Component-Based Distributed Software Reconfiguration: a Verification-Oriented Survey

Hélène Coullon, Ludovic Henrio, Frédéric Loulergue, Simon Robillard

▶ **To cite this version:**

Hélène Coullon, Ludovic Henrio, Frédéric Loulergue, Simon Robillard. Component-Based Distributed Software Reconfiguration: a Verification-Oriented Survey. ACM Computing Surveys, 2024, 56 (1), pp.1-37. 10.1145/3595376 . hal-04067909

HAL Id: hal-04067909
https://inria.hal.science/hal-04067909

Submitted on 6 Nov 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Component-Based Distributed Software Reconfiguration: a Verification-Oriented Survey

HÉLÈNE COULLON, IMT Atlantique, Inria, LS2N, UBL, France
LUDOVIC HENRIO, Univ Lyon, EnsL, UCBL, CNRS, Inria, France
FRÉDÉRIC LOULERGUE, Université d'Orléans, France
SIMON ROBILLARD, LIRMM, CNRS, Université de Montpellier, France

Distributed software built from components has become a mainstay of service-oriented applications, which frequently undergo reconfigurations in order to adapt to changes in their operating environment or their functional requirements. Given the complexity of distributed software and the adverse effects of incorrect reconfigurations, a suitable methodology is needed to ensure the correctness of reconfigurations in component-based systems. This survey gives the reader a global perspective over existing formal techniques that pursue this goal. It distinguishes different ways in which formal methods can improve the reliability of reconfigurations, and lists techniques that contribute to solving each of these particular scientific challenges.

Additional Key Words and Phrases: Reconfiguration, software adaptation, component-based software engineering, formal methods, verification

ACM Reference Format:
Hélène Coullon, Ludovic Henrio, Frédéric Loulergue, and Simon Robillard. 2023. Component-Based Distributed Software Reconfiguration: a Verification-Oriented Survey. *ACM Comput. Surv.* 1, 1 (April 2023), 37 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Through time, distributed software systems have broadly been adopted in industry. Their non-monolithic architectures make development more efficient, particularly when managing large groups of developers with different fields of expertise. They are also more suited to increasingly popular distributed infrastructures such as Cloud computing, Edge computing, or Cyber-Physical Systems (CPS). One of the most relevant examples of current practices is the large adoption of microservice architectures and the extensive usage of Docker containers, which facilitate continuous delivery and continuous deployment [90].

However, advantages of a distributed design come at the price of increased complexity to coordinate different pieces of software, discover and monitor them within the network, deploy them on complex heterogeneous software stacks, and maintain them through time. In particular, the dynamic adaptation of complex large-scale distributed software systems is highly error-prone. For example, a study of 597 unplanned outages that affected popular cloud services between 2009

Authors' addresses: Hélène Coullon, helene.coullon@imt-atlantique.fr, IMT Atlantique, Inria, LS2N, UBL, Nantes, France, F-44307; Ludovic Henrio, ludovic.henrio@cnrs.fr, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, Lyon, France, Frédéric Loulergue, Frederic.Loulergue@univ-orleans.fr, Université d'Orléans, Laboratoire d'Informatique Fondamentale d'Orléans, Orléans, France, F-45067; Simon Robillard, simon.robillard@umontpellier.fr, LIRMM, CNRS, Université de Montpellier, Montpellier, France,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
0360-0300/2023/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and 2015 found that 16% of them could be traced back to a software or hardware upgrade [65]. As more and more critical systems are handled in the Cloud or at the Edge of the network, formal verification of system reconfigurations is more than ever required.

1.1 What is a reconfiguration?

A configuration is characterized by the set of entities that constitute a runtime system, their dependencies, and the runtime setting of these different constituents. Reconfiguring a system consists in changing the configuration while a system is running, i.e., changing the parameters that configure the system, adding or removing entities to the system, or changing the dependencies between these entities. It is generally used to adapt, at runtime, a software system to new running conditions or new requirements.

In a distributed system, configuring a system includes deploying this system, i.e., placing each of the entities 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 optimization or a satisfiability problem [7, 72, 73], while the local configuration often consists in installing and configuring packages and modules, configuring the operating system, and sometimes running containers or virtual machines [54, 59].

This paper focuses on the reconfiguration of distributed software systems deployed on distributed infrastructures. Reconfiguration involves topological changes in the dependencies between system components. In a distributed system, it also includes migrating software entities and reconfiguring host machines. Typical reconfiguration operations include installation/uninstallation/suspension of software entities, connection/disconnection of software entities, migration of entities, substitution of some entities by others, software scaling and elasticity operations.

Of course reconfiguration operations cannot be applied at any given time, the reconfiguration process and its planning are correlated to the internal business behavior of involved entities. For instance, a disconnection should be avoided if an ongoing communication exists between the connected entities. Furthermore, multiple stakeholders (i.e., individuals to whom a specific role is attributed in the reconfiguration process) are involved when reconfiguring: developers, DevOps engineers, system administrators and operators. This makes reconfiguration a complex procedure that needs to be coordinated, planned, and verified.

1.2 Causes of reconfiguration

Reconfiguration is needed for the runtime adaptation of distributed software systems. One of the main reasons for dynamic adaptation are changes in the non-functional requirements (NFR) of an application. Typical examples of non-functional requirements are reliability, security, performance, scalability, or trust. Many of the frameworks we present in this survey have been designed to handle the evolution of non-functional requirements.

However, this survey does not specifically focus on how to fulfill non-functional requirements. This important problem is mostly related to multi-criteria optimization problems, and some surveys dedicated to this topic already exist, in the Cloud or Edge computing for instance [8, 66, 114]. Instead, we provide an overview of methods to guarantee safety properties on the reconfiguration process itself.

Non-functional adaptation procedures must provide a trade-off between different non-functional aspects, e.g., application security, application performance, reactiveness and efficiency of the reconfiguration, etc. The safety of reconfigurations is generally handled independently of these trade-offs. Note however that optimizing the reconfiguration process itself can also have an impact on NFR. For example, it can be difficult to simultaneously achieve performance and safety of the reconfiguration process. Furthermore, both the functional code and the reconfiguration contribute

to the overall non-functional properties of the application, leading to other trade-offs. For example, a slow reconfiguration process will degrade the perceived responsiveness of the application, which might not be acceptable.

1.3 Reconfiguration and component-based modeling

A convenient way to study reconfiguration is to focus on component-based software systems [118]. Software components provide a composition framework with a higher level of abstraction than objects or modules. Components split the application programming into two phases: the writing of basic business code, and the composition phase or assembly phase, which consists in plugging together basic component blocks. Component models[1] provide a structured programming paradigm, and ensure program re-usability. Indeed, in component applications, dependencies are defined together with provided functionalities by means of provide/require ports; this improves the program specification and thus its re-usability. Some component models and their implementations additionally keep an internal representation of the components structure and their dependencies at runtime, allowing introspection of the component structure. Knowing how components are composed and being able to modify this composition at runtime provides great adaptation capabilities, allowing the component system to be reconfigured. We use the term *component* to mean software components, with a very flexible definition: we will apply the term to very structured component systems (e.g., CCM [108], Fractal [31]) as well as classical module systems, or software packages, etc. For instance, we consider both Service-Oriented-Architecture (SOA) and microservices architectures to be instances of component-based distributed software, even though these models adopt a slightly different vision of components compared to usual component models. Notably, SOA adds the notion of contracts, which is not formalized within traditional component models, while in microservices architectures, REST APIs are leveraged as provide/require interfaces and the granularity of services is small so that they can be easily composed.

1.4 Reconfiguration and formal methods

Distributed computer systems are by nature heterogeneous: they can be programmed in different languages, be deployed over varying underlying infrastructures, sometimes relying on different virtualization layers. They are also very often large-scale systems with hundreds or thousands of entities. 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 several actions in parallel is also one of the main reasons to use distributed systems, but it further adds complexity as the system might reach different states depending on the timing of parallel actions. As a consequence, the execution and the reconfiguration of distributed systems is prone to errors. Testing is often inadequate for debugging large-scale reconfigurable distributed systems, as the nature of the components and their composition is not known during development. Additionally, errors often depend on the timing of interactions between entities, or on specific conditions, and are thus unlikely to be discovered by testing.

To ensure the correctness of the reconfiguration process, it is therefore useful to turn to *formal methods* that offer strong generic guarantees about the systems. These methods are based on an abstraction of the system, which is checked against a specification given in a formal language. The granularity of the abstraction may vary, which influences the kind of properties proven and the kind of systems that can be modeled. The main difficulties raised by the use of formal methods are the expertise required to effectively use these methods, and the time needed to define the

[1]Note that the term *model* here refers to the modeling of the software architecture of an application or a system and not to model-driven software engineering (MDE).

right abstraction and/or the correct specification for the system. While the benefit of using formal methods is obvious, the difficulty and the cost required to put them into practice are often prohibitive. This is why several efforts are carried out to provide automatic or easy to use formal tools.

1.5 Scope of the survey and outline

The goal of this survey is to study the solutions that exist to *verify and ensure the correctness of reconfigurations for distributed systems*. More precisely, we will investigate the following research questions on the verification of reconfiguration of distributed software systems:

- **What assumptions on composition models are made in order to verify reconfigurations?** We want to identify assumptions that underlie various models for component-based systems, as they carry over to the verification systems that are based on those models. These assumptions are often shared across multiple models, but are not always stated explicitly, making it difficult to compare the guarantees and assumptions of different works fairly.
- **What properties on reconfiguration are verified and how?** Since reconfiguration affects varied aspects of a system, there exists a wide range of properties to verify. For instance, individual reconfiguration operations or broader reconfiguration protocols may be verified. We classify those properties and the techniques used to verify them.
- **When do verification steps take place, and what expertise is required from stakeholders?** Various verification techniques can be used at different stages during the lifetime of a system, from the design to the execution. Accordingly, the stakeholders involved in the verification vary greatly in role and expertise. We compare the characteristics of verification techniques that make them suitable for different stages and stakeholders of the software life-cycle.
- **What are the open challenges for verified reconfiguration of component-based systems?** The use of formal methods for component-based reconfiguration of systems remains a niche topic. We consider the obstacles that must be overcome to reach wider adoption, in particular from the point of view of programmers and system maintainers; we also sketch potential trends for future research.

The rest of this paper is organized as follows. Section 2 provides a brief overview of different kinds of formal methods. Section 3 describes the concepts of component models. Section 4 lists the four areas of research that structure this survey, as well as one preliminary area of study, each of them explored in the rest of the survey, i.e., in Sections 5, 6, 7, 8, and 9. Section 10 details the positioning of this survey. Finally, Section 11 provides a summary (in particular regarding the above research questions) and discusses open challenges.

2 FORMAL METHODS

Formal methods can be used at different stages of a software system life-cycle, from its conception to its use. This section gives an overview of these stages and the methods that can be used during each of them. We present each method independently, but there is an important trend promoting the combination of techniques for the verification of industrial systems. Some software frameworks embody this trend, such as Frama-C [47, 85] a framework for the analysis and verification of C programs. Note that we omit deductive verification of programs such as Why3 [27], Dafny [100] and VCC [46] that are not used in the contributions studied in this survey.

2.1 Phases of formal verification

Formal specification. Creating a useful and sound system abstraction, and formally specifying requirements over it are non-trivial tasks. They often account for a large proportion of the effort

towards formally verifying a system, in addition to the verification step itself. However, this stage is also the opportunity to carefully plan the system, and avoid many sources of problems for the future. Formal specification constitutes a documentation of the system that is unambiguous and largely machine-checkable. These attributes are desirable for any software artifact, but particularly for one that involves a multitude of stakeholders with different skills and understandings of the system.

Verification at development stage. When developing applications, automatic, interactive, or semi-automatic tools may be used to ensure that the system complies with its requirements. The verification process may target an abstract model, in which case the implementation of the model must be trusted to correspond to the model. For a higher degree of assurance, interactive proof assistants can also be used to write certified programs for the implementation, such that the executable code is extracted directly from the correctness proof [102]. Deductive verification [5, 85] can also be used to verify given programs, and is often more automated than interactive theorem proving. However, to adopt these methods, developers need to understand formal methods and must be able to use the corresponding tools. In addition, interactive and deductive verification are very time consuming. Nevertheless, verification time can be justified for generic applications, algorithms and tools that are massively used in different contexts and difficult to test, such as compilers [82, 101], reconfiguration protocols [28] or reconfiguration languages [64].

Verification at runtime. For formal methods used during the runtime of systems, the emphasis is largely on automated tools that can help detect incorrect system configurations. Generic reasoning tools can be used for these tasks, taking as input a representation of the system in its current state, as well as some (potentially situation-specific) requirements. This declarative approach is easier to justify than a comparable ad hoc procedure, and consequently more trustworthy.

2.2 Techniques for formal verification

Static analysis (SA). Static analysis is a set of techniques that focus on the semantics of programs. The practice originated in compilers, with known analyses such as type checking and variable liveness analysis. In addition, there exist static analysis tools that are independent from compilers, used either to guarantee that programs verify a given semantic property (usually the absence of a class of runtime errors), or to detect bugs. The first category of analyses are often based on abstract interpretation [110] and are sound, i.e., if the tool indicates the property is true, it is guaranteed. An analysis of the second category is both unsound and incomplete, i.e., it detects some potential bugs, but cannot guarantee their absence [56].

Model-checking (MC). Model-checking [17, 44] is a set of techniques to determine whether a high-level model of a system meets a specification. Many of these techniques focus on the verification of properties expressed in temporal logics, allowing the specification of safety and liveness properties. Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) are often used in this context, as well as the more expressive CTL* and μ-calculus. Finite-state automata are the most commonly used class of model for the purpose of model-checking. This representation is useful for distributed systems, because parallel composition of systems is easy to define. Other models, such as Petri nets or process algebras, directly feature a notion of concurrency. Many model-checking tools propose to represent systems through domain-specific languages with specific constructs for concurrency (e.g., message passing) and data types, while they internally rely on more basic models for verification. Most of these models and specification languages can be extended with real-valued time [9, 70] or probabilistic features [81], with corresponding verification algorithms. Model-checking has been successfully used to verify many concurrent systems, algorithms and protocols.

SAT, SMT, and CSP. The Boolean satisfiability problem (SAT) consists in determining whether there exists an assignment of truth values to variables that satisfies a given Boolean formula. Despite the problem being NP-complete, modern provers regularly manage to solve problems with thousands of variables, thanks to a combination of advanced backtracking techniques, efficient data structures and tuned heuristics. This has made them a standard tool to solve, via reduction, many NP-complete problems, in particular those arising in the context of hardware and software verification, for example in the Alloy Analyzer [76]. The Satisfiability Modulo Theory (SMT) problem also consists in determining the satisfiability of a formula, this time in predicate logic with some interpreted symbols (for example, numbers and arithmetic functions are given their usual meaning). Historically, SMT focused only on ground formulas (i.e., those without quantifiers). However, modern solvers also include instantiation mechanisms to deal with quantified formulas, making them able to reason about full first-order logic (FOL) with interpreted functions. Much like SAT solvers, SMT solvers have found many applications in software and hardware verification. Their expressive logic make them particularly suited to data-focused verification tasks. Constraint Satisfaction Problems (CSP) consist in finding an assignment of variables over a finite domain that satisfies a given set of constraints. In some cases, a problem may also require an optimal solution according to a given optimization criterion. Formally, this class of problem is closely related to SAT and SMT. However, research on the topic was historically motivated by applications (in particular, constraint programming [112] and linear programming [51]) whereas the study of SAT and SMT emerged in the context of logic-related inquiries. As a result, the algorithms [30, 95] used for constraint satisfaction differ from those of SAT and SMT solvers, and may be suited to different applications.

Assisted proving (ITP, refinement). In this family, we gather verification techniques that necessitate user input, e.g., proof scripts or program annotations. Interactive theorem provers [24, 106, 119] (ITP) enable their users to define mathematical objects and to prove theorems about them, using highly expressive logical frameworks such as higher-order logic or various flavors of dependent type theory. They can in principle be used to formalize almost any kind of mathematical results, from textbooks to specific software verification problems. Additionally, the validity of user-written proofs is checked by relatively simple algorithms, so that the trusted core of these tools is small. The counterpart to these high levels of expressiveness and trust is that proving theorems is a difficult task that requires familiarity with the logic and tool used, as well as domain-specific knowledge. The proving process is also almost entirely manual, and often time consuming, even for experts.

Refinement techniques proceed by refinements from a high-level specification to executable code, with successive refinement steps that require designing a more detailed model and proving it correct with respect to the model of the previous step. The strength of this approach is that proving a property is typically easier on a high level model than on executable code. Refinement steps are also easy to prove correct, with proofs that can sometimes be automated. In the context of software components, the B method and associated tools are widely used [1, 2], but other approaches exist, for example based on TLA [113] or FDR model-checking tool [74].

Runtime Verification (RV). Since some interesting properties are not decidable, sound static approaches are either automatic but incomplete, or require proofs to be partly written by humans. Another approach is to perform runtime assertion checking [45]. Very often, specification languages for programming languages such as ACSL for C [116], or JML for Java [40], come with tools to translate properties in these specification languages into code that checks the assertions at runtime. In the context of components, properties are sometimes expressed as temporal logic formulas, and the components are distributed: these aspects create additional challenges to tackle [19].

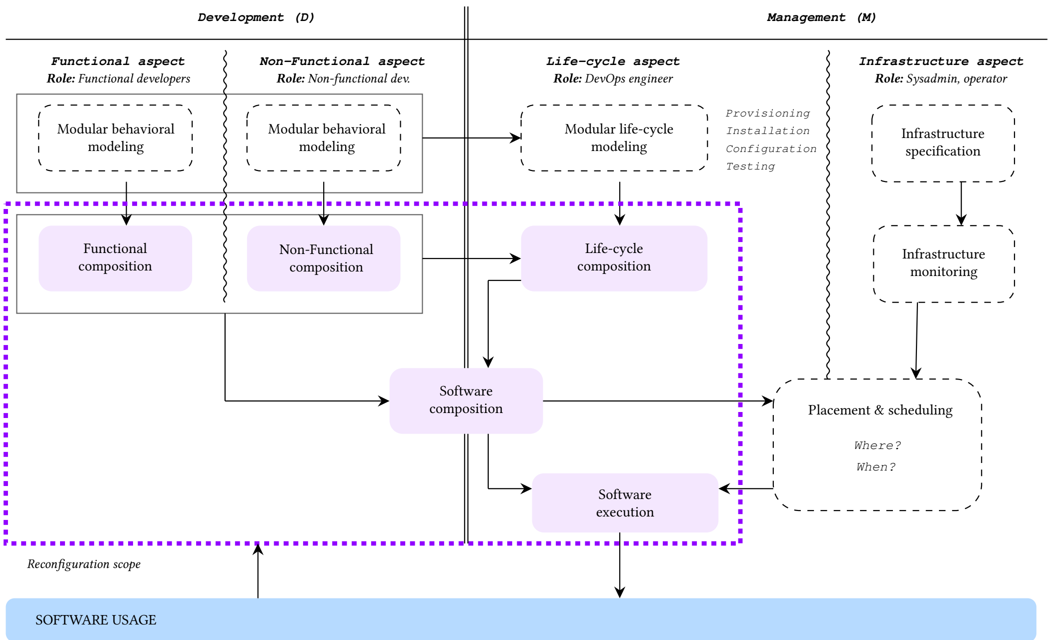

Fig. 1. The workflow when designing a component-based distributed software system. The arrows of the workflow denote temporal dependencies. The workflow is split in two parts: the development (D) and the management (M). Each part is itself subdivided in two different aspects: functional and non-functional; and DevOps and infrastructure. The purple boxes and the purple dashed area are the ones in the scope of a reconfiguration, hence in the scope of our survey, while the white dashed boxes are out of our scope.

3 CHARACTERISTICS OF COMPONENT SYSTEMS

This section introduces and classifies the different component models and their characteristics. This classification is used in Sections 5, 6, 7, 8, and 9 for the purpose of presenting each of the reconfiguration efforts of the literature by stating which component model it targets, and to understand the key characteristics that correspond to this model. This classification is important because a contribution developed with a component model can often be adapted to another one with similar characteristics; this survey aims to identify the key characteristics used by each contribution[2]. Furthermore, this section defines a number of terms used in the survey, summarized in the supplementary online-only material.

Figure 1 illustrates the workflow that is usually followed when designing a component-based distributed software system. It is divided in two parts, development and management, each of which is itself divided into two different aspects that will be discussed in this section. The arrows of the workflow denote temporal dependencies between actions. Finally, the purple boxes and the purple dashed area are the steps considered in the scope of this survey, while white dashed boxes are outside our scope. Each aspect follows a component-based design which splits the development or management into two phases: the design of basic business components or modules, and their composition, or assembly.

[2]See online supplementary material for a table summarizing the terms used in the survey.

3.1 Component structure

Roughly, there are two main ways to organize the components in existing models:

Flat models. Flat models have a single level of composition. Components are peers that can interact with few restrictions. Examples of such models include CCM [108], CCA [23], L^2C [25, 48], BIP [20, 26], Concerto [41], Aeolus [54], Coqcots [32], and actor models [67].

Hierarchical models. Hierarchical models organize components in a tree structure where each component, except the root of the tree, belongs to another component. Each component can interact with its parent component, sub-components, or siblings (i.e., components belonging to the same parent component). Examples of hierarchical component models include Fractal[3] and its implementations [31], GCM [21], SCA [55, 107], and SOFA [34].

In practice, different organizations exist as well. For example, Fractal pushes the concept of hierarchy further than other models, as the interconnections between components can themselves be hierarchical and composed of other components. Another example is bounded hierarchy, as with SOFA, a model that allows instantiating microcomponents (i.e., single simple components) to deal with the management of a given component [34].

Reconfiguring and adapting a large system in a hierarchical way facilitates its compositional management and the scaling up of adaptation procedures [75]. Flat models are generally more flexible but sometimes harder to manage: because many components are at the same level, composing, maintaining or providing reconfiguration capabilities for large systems is often more difficult. In some cases, systems that are flat by nature can be organized hierarchically for some management aspects, e.g., fault tolerance for actors in the Akka framework [125].

Note that another way of handling large sets of components, instead or in addition to hierarchy, is to improve the assembly language with parametrization features (e.g., loops, number of instances of the same component, number of connections). For example, large-scale flat component systems exist: the combination of High Performance Computing (HPC) and component models around L^2C [25, 48] has produced results with a very large number of components.

3.2 Interfaces and connectors

The reference definition of component [118] puts a strong emphasis on component interfaces. They must precisely define requirements of components as well as their provisions, unlike the notion of interface used in object-oriented programming, that only includes functionalities provided by an object. Typically, an input (respectively, output) interface consists of a list of operations that can be used to interact with the component (respectively, that the component can invoke). Interface definition allows compatibility checks during composition (e.g., type compatibility between operations).

In most component models, composition is achieved by interconnecting interfaces through an entity called a connector. Different component models may use different terminologies for interconnection entities. Interfaces are also called ports (some models distinguish ports and interfaces), and connectors are also called bindings or connections. In some component models, the notion of input and output might disappear as a single interface exposes both input and output operations.

The most standard composition model relies on input and output interfaces, and connectors that connects an output interface to an input interface. However, depending on the model, interfaces can be more or less complex, and varying guarantees can be provided when plugging connectors. Most of the advanced models provide extended connectors. One extreme example is the Reo [14] model, whose main focus is on connectors as a way to express computation. In a more standard

[3]In the case of Fractal, with shared components, the component structure is actually a DAG.

way, some component models allow connecting more than two interfaces together and check port compatibility accordingly. In some component models, a notion of cardinality also exists, allowing the port specification to state how many connections can be established from/to a given interface. For example, in Fractal and in many component models, an output port can only be used in one connector to ensure determinacy of communications, but GCM extends the Fractal model with *multicast* output ports that can be plugged to several input ports. L^2C [25] also offers to connect one output port to several input ports. Furthermore L^2C, which is a component model geared toward HPC, offers a specific MPI connector to declare a set of components within the same MPI communicator [117]. With rich interaction patterns comes the definition of protocols implemented by the connectors. Simple examples of interaction protocols include broadcast, lossy channels [123], and arbitrarily complex protocols that can be implemented, e.g., by Fractal's composite bindings.

Interfaces and connectors play a significant role when designing and proving correctness of reconfigurations. Indeed, reconfiguration operations rely on the manipulation of connectors to connect/disconnect components to/from an existing system. For this reason, it is crucial for the definition of reconfiguration procedures that connectors can be observed and modified. Depending on the reconfiguration scenario, it might also be useful to inspect ports to check their characteristics and to ensure compatibility with ports of newly added components.

3.3 Functional and non-functional aspects

The left side of Figure 1 illustrates the *development* part of a component-based distributed software system, itself divided into functional and non-functional aspects. An aspect denotes a cross-cutting concerns related to the development or management of a set of components.

The functional aspects encompasses coding and designing the functional components of the software system, i.e., the pieces of code responsible for fulfilling the business purpose of the application, and their assembly or composition. The non-functional aspect includes writing pieces of code responsible for non-functional components of the software system, i.e., the code responsible for providing all the functionalities that are not the business purpose of the application (e.g., encrypted communications, file format conversion, etc.), and their assembly or composition. Both aspects necessitate two steps: first, the design of basic modules or components; second, the assembly of more complex components (in case of a hierarchical model) by assembling building blocks together and eventually by adding needed *glue* components between them (e.g., for interoperability issues). Both the design of basic components and composite components (i.e., assemblies) can be performed by a large number of developers from different institutions and companies.

In the literature, some component models, such as CCM, CCA, L^2C, BIP or SCA, do not dissociate functional and non-functional aspects of a component-based architecture. Thus, all components of all aspects are designed and connected at the same level and in the same way. As a result, there is no clear decoupling and separation between these aspects. This could lead to difficulties in the design, as both aspects should be considered simultaneously although they involve different experts. Other component models, such as Fractal and GCM, separate these aspects by introducing a notion of *membrane* of a component, which handles non-functional sub-assemblies related to a given component, leaving the functional aspect inside the component.

Note that the term *non-functional aspect* is taken from the literature on component-based software architecture, and is related but not equivalent to the non-functional requirements (NFR) discussed in Section 1.2. NFR are the quality objectives that may trigger a software system reconfiguration, while non-functional aspects of a software system involve the components responsible for non-business code. Reconfiguring either functional or non-functional components of a software system may affect NFR.

3.4 Management and life-cycle specification

The right side of Figure 1 depicts the *management* part of a distributed software system, divided between the life-cycle aspect and the infrastructure aspect. The life-cycle of a component is the specification of the different stages constituting the usage of the component. It typically includes deployment, suspension, update, migration, or deletion phases. The life-cycles of the assembled components also need to interact to avoid errors and disruptions. For instance, a server has to be deployed before a client, or clients have to be suspended before updating a server. For this reason, life-cycles are also subject to composition and this composition is of course tightly coupled to the functional and non-functional assembly of the development part of Figure 1.

Although software engineering has neglected the life-cycle aspect for a long time, many DevOps tools and languages have emerged in recent years that have enhanced the design of management procedures and life-cycles, their maintainability and re-usability compared to ad hoc scripts. For instance, tools such as Ansible, Puppet, Chef, Juju or Kubernetes adopt a component-oriented way of designing the life-cycle and management of distributed software (mainly containerized service- and microservice-oriented software systems). As a result, nowadays, deployment is handled almost exactly as the development aspect: deployment and installations procedures are designed for each component individually, and then composed. This task is typically carried out by DevOps engineers, whose role lie between those of application developers and system administrators.

The life-cycle of a component can be considered as a coarse-grained representation of its behavior. Thus, component models can be characterized by their means of describing the life-cycle of components. At the most basic level, components have only two possible fixed states: they exist or not [20, 107]. This level of granularity is very common, but induces many service interruptions during complex reconfigurations. Consequently, many models also distinguish between *active* and *inactive* components [25, 31]. Finally, other models, specific to the management aspect of component-based architectures (comparable to DevOps tools) offer either advanced fixed life-cycle definitions [59, 60], or the definition of a programmable life-cycle with an arbitrary number of states [41, 42, 54]. Transitions between these states often have to be specified and mapped to actions, for example with a finite state machine [54, 59]. More expressive representations may be used to express parallel transitions inside the component life-cycle [41, 42] (e.g., Petri nets). An alternative to the concept of life-cycle is to consider an infinite number of behavior changes through Dynamic Software Update, where the behavior of the component can dynamically be changed [32]. However, this approach mainly focuses on changing the domain code behavior rather than on the life-cycle aspect.

While the infrastructure aspect is outside the scope of this paper, it is part of the overall management process. It mainly consists of specifying the physical or virtual infrastructure on which the distributed system will be hosted, and monitoring it. When the overall composition of the distributed system and its life-cycle are known, as well as the current configuration of the infrastructure, placement or scheduling problems can be solved to decide where to host which components and which tasks. This aspect is depicted on the far right of Figure 1.

3.5 Configuration and reconfiguration in component models

In component models, a configuration describes both the setup of the components involved in the composition and the structural configuration of the system, i.e., assembling components together. We assume here a distinction between the execution of the application code inside a configuration, and the reconfiguration that consists in modifying the architecture of the application. The boundary between execution and reconfiguration is not strictly identified: some operations, such as modifying some parameters of the components, might influence their behavior in such a way that this may

be either considered as part of the execution or as a reconfiguration action. Here in particular, we adopt a very structural definition of a configuration where reconfiguration means a change in the composition of the component system or the dependencies between components.

Some component models have been designed with reconfiguration in mind, while others have been equipped with reconfiguration capabilities at a later stage, for instance through additional contributions. Some of them handle reconfiguration from the perspective of development, others from a management perspective only, by abstracting the internal behavior of components through their life-cycle. In any case, to handle reconfiguration, a reconfiguration language is required to specify which operations to apply on the current structural assembly of components (i.e., configuration). A reconfiguration language provides at least the following four structural operations: (1) adding a component in the assembly of components; (2) removing a component from the assembly; (3) connecting (i.e., binding), the compatible ports of two components; and (4) disconnecting (i.e., unbinding) the ports of two components. Note that for hierarchical component models, the aforementioned four operations are adapted to composite components [99].

When a component model is equipped with reconfiguration capabilities, the set of available operations form a specific language. This language is either provided as an API of the component model implementation or sometimes as a reconfiguration DSL (domain specific language). In the case of Fractal for example, the component model comes with a reconfiguration API, but a reconfiguration DSL called FScript [99] has been designed afterwards. Each reconfiguration language additionally offers reconfiguration operations specific to the model's way of handling reconfigurations, component life-cycles or component behaviors. For instance, [99] is based on Fractal, and thus offers operations to start or stop components. Coqcots [32], that relies on Dynamic Software Update (DSU) to change the internal behavior of a component, offers the associated reconfiguration operation `hotswap`. Concerto [41, 42] or Aeolus [54] abstract the internal behavior of components through programmable fine-grained life-cycle, their reconfiguration languages respectively offer the operations `pushB` to apply a specific subset of the life-cycle, and `stateChange` to move from one step to another in the life-cycle. The reconfiguration language of Reo [14] stands apart: although it offers operations for structural modifications that are analogous to the other models, these operations target connectors instead of components. Specifically, reconfigurations in Reo modify the internal structure of connectors, which governs their behavioral semantics.

The stakeholders involved in a reconfiguration process are potentially the same as those involved in the development and management parts. In Figure 1, the four types of stakeholders (i.e., roles) of a reconfiguration are depicted: developers, responsible for designing and implementing the functional and non-functional aspects of modules or components; system administrators and operators, responsible for maintaining, configuring, and testing multi-users computer systems such as servers or Clouds; and, in between, DevOps engineers, responsible for procedures and workflows for deployment, integration and management of distributed systems within distributed infrastructures. These stakeholders may have very different skills and types of expertise. Hence, separation of concerns between roles is important when reconfiguring a running system.

3.6 Discussion

Component models help structure an application's code in distinct sub-functionalities that communicate, and allow clear separation of concerns between the development of domain-specific code and the assembly of components as black boxes through their interfaces. Thus, components facilitate the programming of distributed software, and the action of choosing dynamically the components that participate in an assembly (i.e., reconfiguration). Some component models have been equipped with reconfiguration capabilities that allow the component assembly to be adapted at runtime; they are discussed in the rest of this survey. In all contributions studied in this survey,

	structure	*interfaces*	*aspects*	*reconf. op.*	*life-cycle*	*formal spec.*
Fractal [31]	hierarchical	programmable*	functional non-functional	✓	fixed life-cycle*	mechanized (Alloy)
GCM [21]	hierarchical	multiplicity	functional non-functional	✓	fixed life-cycle	mechanized (Isabelle, Coq)
L^2C [25]	flat	multiplicity	functional	✓	fixed life-cycle	manual
BIP [20]	flat	multiplicity	functional	none	on-off	manual
Aeolus [54]	flat	multiplicity	life-cycle	✓	programmable	manual
Concerto [41]	flat	multiplicity	life-cycle	✓ + `pushB` + `wait`	programmable	manual
Reo [14]	NA	programmable	non-functional*	connector reconf.	no*	manual
Coqcots [32]	flat	multiplicity	functional	✓ + `hotSwap`	DSU	mechanized (Coq)

✓: the model offers the four usual topological operations (add/remove/bind/unbind)

Table 1. A classification of the component models mentioned in this survey. Cells marked with a '*' are further discussed when commenting the figure.

the reconfiguration is considered at the composition level (i.e., architecture, assembly). The purple dashed rectangle of Figure 1 indicates this scope. In particular, dynamically changing the domain code of components is out of scope (see Section **??** for a discussion on related contributions outside the scope of the survey).

Table 1 presents the characteristics for the component models that will be cited in this survey. These characteristics, as discussed above, are the following:

- *structure* = { flat, hierarchical };
- *interfaces* = { simple, multiplicity, programmable };
- *aspects* = { functional, non-functional, life-cycle};
- *life-cycle* = { on-off, fixed, programmable, DSU }.

Note that by "multiplicity" of interface we mean properties such as connecting multiple components to a single interface, or the cardinality mentioned in Section 3.2.

We additionally mention whether any formal specification of the component model has been published and with which formal method or tool, if applicable. The table is not meant to be exhaustive. Indeed, some component models have been excluded from this survey because they possess no reconfiguration capabilities, or no formal specification. This is the case of well-known component models CCA [23], CCM [108, 109], and SCA/Frascati [115].

A few elements of this table, marked with a star, have to be discussed in more details. First, in the Fractal component model, the membrane can be used to intercept messages entering or exiting a component, allowing some form of programmability of the interfaces; also an interface can be a *collection* of interfaces, allowing the instantiation of several identical interfaces. For this reason interfaces in Fractal are considered programmable. The life-cycle of Fractal components is not exactly fixed but it is predefined and can be programmed. However there is no specific support for the programming of the component life-cycle. Second, Reo is a unique model, as it primarily describes connectors rather than the components themselves. It mainly focuses on the non-functional aspect of distributed software (i.e., their communications and interactions), without

handling the domain code. Hence our classification for structure is not applicable to this model. Furthermore, Reo has no concept of life-cycle for the connectors. Third, L^2C is a component model made for high performance computing that is rather different from others. Nonetheless, it has been formally defined [97], and extended with reconfiguration capabilities, in particular for dynamic numerical simulation codes [98].

4 CLASSIFICATION OF CONTRIBUTIONS AND ORGANIZATION OF THE SURVEY

In this section, we identify areas of research (AR) related to verified reconfiguration. These areas of research are used in the survey to classify the existing literature.

Surveyed contributions typically involve four dimensions: (1) a component model; (2) a verification technique; (3) a verified property; and (4) an application or algorithm for which the property is ensured. Most presented papers contribute on one of these four aspects, but in a context that covers the others as well. The classification of component models (Section 3 and preliminary study below) and formal methods (Section 2) is used to compare the contributions. Note that the verified properties are strongly linked to the different AR. Presented contributions sometimes use an application as an illustration of their approach, with the idea that it should be applicable to any application or system.

Applicability of contributions are another important dimension. While researchers are interested in the details of the specifications, programmers are more interested in the easy applicability of the contributions. For this reason, this survey identifies the level of formal methods expertise that is required to apply the technique of each contribution. The classification of component models also allows users to identify the kind of systems for which the presented technique is applicable.

This survey is thus organized as follows:

Preliminary Study: Formalization and verification of reconfigurable components.

This area, which could be the subject of a dedicated survey, covers formal component models and their verification. Although it does not deal directly with reconfiguration, this area is relevant because it forms a basis for verified reconfiguration. Verified properties apply on the model itself, and thus hold for any component configuration. Component models that handle reconfiguration usually provide a reconfiguration language offering a set of operations. Hence, this area of research also explores formal models and verification applied at the reconfiguration operation level.

AR1: Verification of configurations.

Unlike the preliminary study, this topic is specific to a given application or system instance (i.e., a given configuration). For a given application or system written with a component and reconfiguration model, and a given configuration, the research in this area investigates the kind of properties that can be verified statically. This question can also be extended by considering whether some properties stay true from one configuration to another, or whether they can be re-verified after a reconfiguration.

AR2: Preparation of a reconfiguration.

While the previous area focuses on static verification of a given configuration (e.g., structural properties), this topic includes the consistency of the internal state of an application (e.g., behavioral properties), and how to prepare this internal state so that the system can be reconfigured safely. A related topic is also how to prepare the infrastructure on which the system is deployed to ensure properties during the reconfiguration.

AR3: Verification of a reconfiguration program.

Once the source configuration has been verified and prepared for reconfiguration, and once the target configuration has been chosen, a reconfiguration script or program is needed to move from

the source configuration to the target configuration. This program must satisfy some properties, such as correctness, safety, or performance. A script can be verified a posteriori, or be automatically synthesized in a way that ensures correctness by design.

AR4: Inference of target configuration.
This area of research involves techniques to infer target configurations that guarantee some properties. Since a reconfiguration is a transition from an initial configuration to a target configuration, it is highly relevant to ensure that a correct target configuration is chosen.

An illustration of the areas of research. In order to delineate these different research areas, let us consider the example of scaling a web service based on SCA. In this example, the preliminary study considers the verification of properties guaranteed by the underlying component model, namely SCA. AR1 focuses on the verification of properties of the application before or after the scaling operation, to ensure that the service is functioning as intended. AR2 is the topic of leading the web service to a state where it can be scaled (e.g., pausing and backing up ongoing transactions). AR3 is the verification of the scaling operation itself, either by checking that the program used to execute it satisfies the required properties, or by generating such a correct program automatically. Finally, AR4 is the task of determining, whenever a scaling operation is initiated, what services should be deployed or stopped (e.g., to meet quality of service and non-functional requirements).

Contributions to this preliminary study and these areas of research are detailed in Sections 5, 6, 7, 8, and 9, respectively. Citations corresponding to the contributions in each section are indicated by boldface. A comparison of the contributions is given in a table at the end of each section, based on the six following criteria (when applicable):

1. *reconf. aspects*: which aspects are reconfigured by each contribution, i.e., functional (*func*), non-functional (*nfunc*), or life-cycle (*lifec*), and therefore which stakeholders are involved in the reconfiguration (see Figure 1);
2. *reconf. op.*: which reconfiguration operations are handled by each contribution (✓denotes the four basics architectural operations);
3. *properties*: which category of formal properties are verified by each contribution, i.e., structural (*struct*), behavioral (*bhv*), correct-by-design (*CBD*);
4. *verif. technique*: which verification techniques and tools are used to verify those properties (see Section 2);
5. *expertise*: which level of expertise in the field of formal methods is required from the programmers: *none* when the verification process is automated, *DSL* when a higher abstraction level language or tool is offered to the stakeholders to easily write properties and/or perform verification, or *FM* when the technique requires knowledge of formal methods; the table description further indicates which stakeholders are concerned with the area of research;
6. *comp. model*: which component model, if applicable, is used by the contribution (see Table 1 and Section 3).

In each table, contributions that are placed in a very similar setting are gathered on the same line, e.g., if one article is the follow-up of another one.

5 PRELIMINARY STUDY: FORMALIZATION AND VERIFICATION OF RECONFIGURABLE COMPONENTS

Various formalizations of component models exist in the literature, either because different composition models require different formal models, or because the various properties of interest for a given component model lead to various formalizations.

Already, the different composition models that exist (as described in Section 3) tend to favor differences in formalization. Indeed, formalizing a model such as Reo [80], which focuses on connectors, cannot be done in the same way as the formalization of Fractal **[104]** where an application is mostly described by business code embedded in components, or in the same way as BIP **[20]**, which provides a simple but complete model encompassing the specification of both business code and interactions. Reo is also a typical example of the different formalisms that can exist for a single component model: [80] lists as many as thirty of them including co-algebraic semantics [16], constraint automata [18], or tile model [15]. However the tile model is one of the only semantic models built with reconfiguration operations in mind.

This survey is not intended to be a detailed comparison of the different formalizations, but it is important to remark that the formal definition of a component model is a prerequisite to the formal verification of component-based systems and of their reconfigurations.

Formalization methodology. There is no common view on how to formalize a component model. Consequently, some works and definitions are difficult to relate precisely. For example, one-to-one communication is very common in component models but the restriction that a port that emits one-to-one communications cannot be bound to more than one port is formalized differently in each formal model. Besides a formal model, verification typically requires a specification language to express properties that are meant to be verified. The choice of specification language therefore depends on the model as well as the type of properties to be verified. One of the most common type of property to check are invariants, including topological properties on the assembly and relations between values of systems parameters. Many such invariants can be formulated implicitly in the model, for example by enforcing a type discipline on components **[32]**, but others require an explicit formulation. In order to describe more finely the functionalities of the system and its components, one often needs to specify properties of inputs and outputs: in most cases, this requires a logic with quantification over values **[104]**. Lastly, one may wish to specify the expected evolution of a system, including termination and liveness. To express such properties, the specification must allow the expression of temporal logic formula **[12]**.

An illustration of different approaches to formalization. Consider for example the Fractal component model: the formalization of the model in Alloy defines a strict encoding of the specification and an axiomatic characterization of the model **[104]**. Other formalizations adopt a formal model structure close to the component structure and integrate the component hierarchy and the relationship between the different entities **[12, 68]**. For example, the axiomatic definitions include a relation (in the mathematical sense) between ports and the component they belong to, whereas in structural formalizations a port is an element of the component structure: the axiomatic formalization is more declarative but less structured[4].

Somewhat similar to the work on Alloy is the formalization of GCM component configuration **[63]** and reconfiguration **[64]** in Coq. Like the Alloy specification, the functional behavior is not taken into account, but in **[64]** runtime reconfiguration is formally specified and the user of the platform can prove properties on the architecture of the application at runtime.

What is formalized? What properties are proven? Another important point of distinction between different formalizations is the way operational semantics (i.e., the way the component model describes the execution of the system) is taken into account. Many formal models focus only on structural aspects **[104]** but some take the behavior of the component into account **[12, 20, 68]**. The relation between the structure of the components and their behaviors then allows provers to

[4]See online supplementary material for a more detailed example.

reason about possible events that can occur at runtime. For example, two components that have no ports connected to each other cannot interact.

It is useful to distinguish generic requirements – that always apply to the model – from those that are specific to an application. In the latter case, the stakeholders who formulate these requirements may not be familiar with formal methods, and typically require a specification language that is tailored to the domain. For instance, BIP components provide correctness-by-construction, by code generation from a model, and ensure generic properties that are valid for any configuration of any BIP application. Thus, a programmer only has to ensure that an application is programmed correctly to ensure that this application satisfies BIP properties **[20]**, even when the BIP program is the coordination of actor communications **[26]**. It is worth noting that, unless there is a formal verification effort on the tool itself, there are no strong guarantees on the correctness of the code or models generated by the tools. Work on testing C compilers shows that such tools often contain numerous bugs but that formally verified ones do not [126].

Formalization and verification of reconfiguration operations. Recall that reconfiguration operations are the means provided by the component model to change the component assembly at runtime. Most of the formalized component models that offer a basic structural reconfiguration language also formalize the four associated structural operations (bind/unbind/add/remove) and their semantics on the assembly of components, sometimes focusing on some of the operations. One could distinguish works that prove structural properties from works that prove correctness of the operations w.r.t. the component behavior. A typical example of the first category is the proof of structural invariants on a Fractal component hierarchy **[99]**. In this work, through introspection and specification of the effects of operations, the authors are able to verify structural properties on a reconfigurable component assembly, using Alloy. Coqcots **[32]** reaches a higher degree of specificity through a formalization of reconfiguration operations using Coq. Finally, pre- and post-conditions can be added to reconfiguration operations to guarantee more structural properties **[32, 99]**; in this case, pre- and post-conditions are part of the specification of the reconfiguration operation, but their verification can only be checked on a given configuration[5].

Reasoning at the same time about the behavior of a component assembly and its reconfiguration operations is a challenging task and has only be achieved in restricted cases. One solution is to reason on an abstraction of the behavior. For example, Concerto [41, 42] and Aeolus **[54]** abstract the internal behavior of components through programmable life-cycles, and respectively offer in their reconfiguration language the operations `pushB` to execute a specifc subset of the life-cycle, and `stateChange` to move from one place to another in the life-cycle etc. In the domain of theorem proving, **[68]** uses Isabelle/HOL to prove a theorem on the replacement of a component and its effect of the internal semantics of the component; this can also be seen as an abstraction of the component behavior, but from a different perspective. Reasoning on more complete behaviors and reconfiguration seems achievable but requires a significant formalization effort. For example, [11] is able to verify a rich application with a complex behavior, but focuses on the binding of new components to provide redundancy in a fault-tolerant application.

Reconfiguration in a given model may also be considered as a decision problem, which amounts to determining whether a reconfiguration exists to go from one given configuration to another. Decidability and complexity results were given for various restrictions of the Aeolus model **[54]**.

Table 2 summarizes the contributions presented in this section, according to five of the six metrics introduced in Section 4. The metric *expertise* has been omitted because verification at the level of the component model does not require any expertise from the user, except knowledge of the

[5]see online supplementary material for an example of a structural specification of an *add* operation in Alloy.

	Reconf. aspect	*Reconf op.*	*Properties*	*Verif. technique*	*Comp. model*
[32]	func	✓ + `hotSwap`	struct *soundness*	ITP (Coq)	Coqcots
[99]	func nfunc	✓	bhv *pre- and post-conditions*	SAT (Alloy)	Fractal
[104]	func nfunc	✓	struct *soundness*	SAT (Alloy)	Fractal
[68]	func nfunc	✓	struct + bhv *soundness*	ITP (Isabelle/HOL)	GCM
[12]	func	✓	struct + bhv *soundness*	paper proof	GCM
[63, 64]	func nfunc	✓	bhv *soundness*	ITP (Coq)	GCM
[20, 26]	✗	✗	bhv *soundness*	paper proof	BIP
[54]	lifec	✓ + `stateChange`	bhv *decidability*	paper proof	Aeolus

✓: yes with the four usual topological operations (add/remove/bind/unbind),
✓: possible with the model but not provided, ✗: no

Table 2. Comparison of the contributions to the preliminary study according to the criteria established in Section 4. The criterion *expertise* is not applicable to this area of research.

component model itself. Each contribution that verifies some properties on the component model is represented in this table. For instance, previously mentioned contributions about Concerto do not provide any verification at the model level, and are thus not present in the table. Note that we distinguish in this table the models for which providing reconfiguration operations would require a deep modification of the model (✗), from those that already expose enough features to make such operations possible (✓). The formalization of a component model is not only useful to reason about the behavior of applications. It is also necessary to prove the correctness of protocols that manipulate components, e.g., to stop or reconfigure them. Those aspects are thus central to many of the works presented in this survey, even though the precision and rigor of the formalization of the component model vary. To summarize, the formalization of a component model is a prerequisite for the proof of correctness of reconfiguration. As often in formal methods, the careful design of a model is crucial to later write effective proofs. However, to the best of our knowledge, no unification framework and no precise comparison of different formalizations of component models exist yet.

6 AR1: VERIFICATION OF CONFIGURATIONS

In this section, we explore efforts to verify properties of a component-based system in one given configuration. In other words, these contribution address reasoning about the behavior of a component system that does not change its structure. While this area of research is not strictly related to reconfiguration, we focus our attention toward contributions that target settings where reconfiguration is available. In this respect, we distinguish three types of contributions: work that serves as a

foundation for further efforts on verified reconfiguration, techniques to ensure that some properties remain valid after a reconfiguration, and techniques to (re-)verify properties after a reconfiguration.

The first category is by far the broadest, as verifying a single configuration is a necessary step before being able to verify reconfigurations [33, 39, 77]. Somewhat unintuitively, among the models and verification techniques that focus on systems in a given configuration, many deal primarily with temporal properties. Indeed, a system configuration does not correspond to a static state: the system is still subject to internal evolutions and events at a finer scale. Reasoning about the functional behavior of a system between reconfigurations is often achieved through automata-based representations such as *parameterized networks of synchronized automata* (pNets) for GCM **[12]** and *constraint automata* for Reo [18], or through process algebraic descriptions such as the `mCRL2` specification language **[87]**.

Corresponding model-checking techniques can be used to verify properties of these representations **[11]**. In this context, reconfigurations are operations that modify the automaton itself, and typically require dedicated techniques to be reasoned about. Since a posteriori analysis of automata-based models can be challenging, some approaches rely on properties of the model itself to ensure correctness. For example, BIP **[20]** uses model-guaranteed properties and behavior-preserving transformations to ensure correctness by construction. Other approaches can be used to specify and verify the behavior of components in a given state and their interactions, such as session types [121] or model-theoretic semantics [22].

Let us now consider verification techniques that use information about the current configuration to ensures that some properties are still valid after reconfiguration. Such techniques are especially useful in the planning phase before a reconfiguration. To this end, Lanoix et al. define a notion of *component substitutability* **[96]** in the context of a hierarchical component model. This simulation relation denotes whether substituting a (set of) component(s) by another will preserve the behavior of the system. This notion of substitutability is closely linked to refinement, and indeed this work relies on B tools to check substitutability constraints. Other contributions divide the verification of reconfigurations in two tasks: verifying some properties of configurations, then verifying that those properties are preserved by reconfigurations. This is the approach of **[99]**, which defines a precise notion of system consistency for configurations of Fractal components. This definition is partly application-specific, and an instance of the definition needs to be specified, then verified with the help of the Alloy Analyzer. That paper also describes an approach to verify the preservation of consistency by reconfigurations (see Section 8).

Lastly, we discuss verification techniques used after a reconfiguration or a series of reconfigurations. Here the challenge is to leverage prior knowledge to facilitate re-verification. The component-based approach enables verifiers to isolate components that have been affected by a reconfiguration, which in the context of hierarchical component systems includes both the components that are directly targetted by the reconfiguration as well as the composite components that contain them. In **[79]** and **[78]**, these systems are represented through term algebras, with reconfigurations corresponding to term substitutions. This representation is used to identify parts of a system affected by a reconfiguration and to keep track of dependencies. It serves as the basis for compositional and incremental verification techniques, minimizing the number of verification tasks to perform upon reconfiguration. In **[79]**, this idea is used in the context of model-checking, specifically targetting probabilistic properties (e.g., the probability of a component being in a given state, which may depend on sub-components). This work is extended in **[78]**, this time targetting verification with SMT solvers.

Table 3 summarizes the contributions presented in this section, relating them to the criteria presented in Section 4. The focus of the articles presented in this section is not on reconfiguration,

	Reconf. aspect	*Reconf op.*	*properties*	*verif. technique*	*expertise*	*comp. model*
[11, 12]	func	✓ bind in **[11]**	bhv *any temporal property*	MC (CADP)	FM (temporal logic)	GCM
[87]	✗	✗	bhv *safety*	MC (CADP)	FM (temporal logic)	Reo
[96]	func nfunc	✓ + substitution	bhv *substitutability*	refinement (B method)	FM (B method)	Fractal
[20]	✗	✗	CBD	SA	none	BIP
[99]	func nfunc	✓	struct *invariants*	SAT (Alloy)	FM (Boolean logic)	Fractal
[79]	nfunc	substitution	struct *probabilistic*	probabilistic MC (PRISM)	FM (temporal logic)	ad hoc hierarchical
[78]	nfunc	substitution	struct *invariants*	SMT (Z3)	FM (FO constraints)	ad hoc hierarchical

✓: yes with the four usual topological operations (add/remove/bind/unbind),
✓: possible with the model but not provided, ✗: no

Table 3. Comparison of the contributions to AR1 according to criteria established in Section 4.

thus there are not many reconfiguration operations verified in these works. However, some papers relate their contribution to reconfiguration operations, such as the substitutability criteria in **[96]**. Also, some contributions focus on functional verification but are able to verify some reconfiguration operations, e.g., the binding of some components in a fault-tolerant application **[11]**. The properties verified in this area of research are typically specific to an application, and should be specified by stakeholders at this level (e.g., DevOps engineers), yet knowledge of formal method is usually required to specify these properties. The BIP component model is an exception, as it provides correct-by-construction guarantees **[20]**, however it has little support for reconfiguration. The specification of Fractal in Alloy also plays a particular role as it takes reconfiguration into account and guarantees invariants on configurations, but the required expertise is quite high (Alloy framework) and the guaranteed properties only consider the structure of components, rather than their behavior.

By nature, some contributions, such as the ones related to Reo, only consider non-functional properties of systems or reconfigurations **[78, 79, 87]** while other are inclined towards functional properties without excluding non-functional properties. In particular in the Fractal and GCM cases **[12, 99]**, the membrane handles non-functional sub-components related to a component. When the formal model includes the formalization of membranes, non-functional properties of the assemblies and reconfigurations can be considered [94].

7 AR2: PREPARATION OF A RECONFIGURATION

To alleviate the complexity of dynamic reconfiguration, most approaches rely on a clear separation of concerns between application-specific behavior and architectural aspects on which the reconfiguration is performed. Many contributions in the literature assume this separation, and focus on the problem from the perspective of software architecture, not taking into account the application-specific states of the system.

A few papers try to characterize the application behavior in an abstract way, in order to analyze the conditions under which a clear separation of concerns between the application-specific aspects and the reconfiguration aspects can be achieved. Other approaches, although they abstract away

the application behavior, offer a fine-grained modeling of component life-cycles with user-defined finite state machines. There are therefore many possible states for a system, even without considering application-specific states. In both categories of approaches, a fundamental question is to characterize, and then provide ways to reach, states of the system where a reconfiguration can be executed in a safe manner.

In **[89]**, an application structure is formalized as a directed graph. The four usual reconfiguration operations are considered. The authors develop a notion of application state, with the goal of making the contribution independent of any specific application. The system component that is responsible for the dynamic configuration changes, called the configuration management, needs to direct the part of the application to reconfigure towards a *quiescent* state, and to check that the application is in such a state. The application behavior is abstracted as follows: nodes initiate, accept and service transactions that are basically directed point-to-point communications between two nodes. The life-cycle of a created node has two states, passive and active. In passive state, a node is not engaged in a transaction it initiated and cannot initiate new transactions, but it should accept and service incoming transactions. A quiescent state is a passive state with the additional conditions that it is not servicing a transaction, and will not accept incoming transactions. The reconfiguration operations have pre-conditions similar to those found in many other works (e.g., a node cannot be removed while linked) but also include conditions on the application state via the notion of quiescent state. For independent transactions and bounded time transactions, the paper shows that a node can always reach a quiescent state in bounded time. This is further extended to dependent transactions, and a hierarchical model of components, with some restrictions concerning transactions at the upper level. The component model shares many similarities with Fractal.

Vandewoude et al. **[122]** propose an alternative to quiescence that they name *tranquility*. The authors assert that the requirements for quiescence make dynamic updates based on this concept potentially disruptive. In this new framework, a node is tranquil if it is not involved in a transaction it began and will not initiate new transactions, it is not servicing a request, and its direct neighbors are not engaged in a transaction that has or will require its participation. Updates based on tranquility do not require other nodes to be passivated. It means that a node may never reach a tranquil state, and also that it should be blocked when it reaches this state in order to be updated. Therefore tranquility may not be reached in bounded time and the authors suggest to use quiescence as a fallback mechanism in systems that implement tranquility to ensure that updates are executed in bounded time. Moreover, the approach relies on the strong condition that "there may exist no dependencies between nodes that are not connected to one another". This means in particular that sub-transactions are independent from the original transaction. These sub-transactions can be served by the new version of a component, but this is very restrictive in practice. This methodology is applied to the Draco component framework, which share similarities with GCM except that the model is flat. In particular, it relies on asynchronous messages passed along connectors and has an implementation based on Java classes, like GCM-ProActive.

As for hierarchical systems, Henrio and Rivera **[69]** propose an algorithm to reach a quiescent state for a composite component and all its subcomponents. This algorithm stops a set of GCM components that communicate by a request-reply mechanism and FIFO queues. They propose an algorithm in two phases that is specified semi-formally[6]. The algorithm never deadlocks but may not terminate if the component connectors create loops. The authors conclude by stating that their "major objective is to verify formally the stopping algorithm, but the complexity of the algorithm is a hard obstacle." The Algorithm illustrates the complexity of the specification

[6] see online supplementary material for more details.

and the verification of reconfiguration algorithms in complex component systems with powerful communication mechanisms between components.

Ma et al. **[103]** improve both quiescence and tranquility. The authors share the view of Vanderwoude et al. that quiescence leads to too much disruption in the dynamic update of distributed systems. Moreover, they point out the limitations of the hypothesis of tranquility. Both quiescence and tranquility are local properties, i.e., properties of a node and close neighborhood. The author propose a global property named *version consistency* that applies to all transactions of a system with respect to a dynamic update. A necessary local condition called *freeness* ensures version consistency. Simulations show that two algorithms to achieve freeness are more efficient (better timeliness and less disruption) than the approach of Kramer et al. in addition to being suited to a more liberal definition of transaction than tranquility.

These three approaches are rigorous in the sense that they provide precise definitions of the concept they introduce, but only proof sketches are provided, and proposed algorithms and protocols are not proven correct.

Lanore et al. **[98]** target reconfigurable HPC applications. This work presents the formalized component model DirectMod, which extends L^2C with reconfiguration features. In DirectMod, an assembly of components is divided into reconfigurable sub-assemblies, called *domains*. More than one domain can be specified in an assembly, which makes concurrent reconfigurations possible. However, in DirectMod the specification of the quiescent state of a domain is left to the developer, which is a difficult and error-prone task. It consists in locking/stopping the domain, thus isolating it from the rest of the assembly, without introducing deadlocks in the overall code.

Concerto **[42]** is a recent component-based deployment model with reconfiguration capabilities, inspired by Aeolus [54]. Concerto relies on the concept of component behaviors that are defined by the component developer. For instance, a behavior could be defined to include the actions required to deploy the component. In addition to the usual four reconfiguration operations, Concerto offers an instruction to queue a request for a component to execute a given behavior. The execution of this instruction is non-blocking and introduces asynchronicity in the reconfiguration, while a synchronization command is provided to wait for a component to execute all its behavior requests. A component can only change its behavior when it has executed all actions associated with its current behavior, which can be compared to a safe state or quiescent state[7]. For other operations, typical conditions apply, e.g., a component can be removed only if its port are no longer connected.

Table 4 classifies the five main contributions discussed in this section according to the six criteria introduced in Section 4. The contributions about quiescence and related concepts **[89, 98, 103, 122]** consider the usual topological operations and, as they are concerned with the interaction between the application behavior and the reconfiguration behavior, specific operations such as activate and passivate are also offered. In several approaches, these operations may be not directly available to the developer: in this case, a higher-level “component update” operation is provided. Concerto focuses on the life-cycle aspect and its set of operations includes two specific reconfiguration operations. All these contributions define a notion of state in which a reconfiguration can happen, and prove on paper that such states are reachable. Implementations or algorithms are provided to reach such states, thus providing a basis to develop automated systems. Verification techniques play as different role in this area of research, as the verified aspect is often a generic algorithm that allows any application to reach a quiescent state. Consequently, the programmer that calls the reconfiguration preparation algorithm is in general not exposed to the formal methods used to prove the generic algorithm, and no user expertise is required.

[7]see online supplementary material for an illustration of a concerto system.

	Reconf. aspect	*Reconf op.*	*properties*	*verif. technique*	*expertise*	*comp. model*
[89]	func nfunc	✓ + `activate` + `passivate`	bhv *reachability*	paper proof	none	Fractal-like
[122]	func nfunc	component update	bhv *reachability*	paper proof	none	Draco: similar to GCM but flat
[69]	func nfunc	stopping signals	bhv *progress*	paper proof	none	GCM
[103]	func nfunc	component update	bhv *necessary condition*	paper proof	none	ad-hoc flat with ports and connectors
[42]	lifec	✓ + `pushB` + `wait`	–	operational semantics	–	Concerto
[98]	func	✓ *graph transfo*	–	operational semantics	–	L^2C

✓: yes with the four usual topological operations (add/remove/bind/unbind)

Table 4. Comparison of the the contributions to AR2 according to the criteria established in Section 4.

The goal of the contributions related to quiescence **[89, 98, 103, 122]** is to safely decouple the business logic part of distributed applications from the reconfiguration part. Such decoupling makes the reconfiguration of both functional and non-functional aspects safer. In the case of Concerto **[42]**, non-functional aspects of reconfigurations are target of analyses, in particular execution time [41].

8 AR3: VERIFICATION OF A RECONFIGURATION PROGRAM

The topic of program verification, i.e., verifying whether a program satisfies some correctness property, is central to the field of formal methods. Various properties can be used to characterize the correctness of this reconfiguration, including invariants, post-conditions, liveness properties and more. Generic verification techniques require a logical framework that is able to accurately represent the semantics of reconfigurations as well as to express a wide range of properties over them. Naturally, verification techniques vary greatly depending on the logical frameworks[8].

Model-checking over automata-based representations of systems is among the most common and successful of these techniques. For example, the semantics of Reo connectors can be described through *constraint automata* [18], a formalism based on a labeled transition system. Note that a constraint automaton describes the behavior of a Reo connector in a given state, not the evolution of such a connector during a reconfiguration. Nevertheless, constraint automata can be used to reason about connectors between reconfigurations. This is the approach of ReCTL* **[43]**, a branching-time logic with modalities denoting reconfigurations. In a closely related work, Krause et al. introduce another automaton-based semantic model named *distributed constraint automata with state memory* **[91]** to bridge the gap between low-level semantics of Reo and high-level representations of reconfigurations based on graph transformations [86, 92]. Model checking was also used to verify some properties of deployments in a restricted version of the Concerto model **[49]**. In this work, besides typical correctness properties, quantitative properties of the execution are also

[8] see online supplementary material for an example of property specification and reconfiguration program, in Concerto.

symbolically checked in order to assist in the development of efficient deployments. Lastly, Gaspar et al. **[62]** use *pNets* to model an industrial application. While the verification of the static version of the application is possible, the verification of a version with dynamic reconfiguration leads to state-space explosion, and only some properties of interest can be verified. In this article, the CADP framework is used to verify properties expressed as temporal logic formulas. For example, the authors check that "there is no path in which the `flag` evaluates to false without the occurrence of a `Q_HMStopMethod`."

Besides automata-based model checking, a deductive approach can also be used for the verification of reconfigurations. A resource logic, inspired by separation logic and used to specify configurations, is presented in **[6]**, along with a Hoare-style calculus to verify reconfiguration programs. Related complexity and decidability results are given in **[29]**. The component model used in these works adopts a very logical perspective above interactions-based component design *à la* BIP (the authors also claim some similarities with Reo).

To use notions of pre- and post-conditions for reconfiguration while avoiding the complexity of Hoare-style deductive verification, one solution is to take a transactional approach, as is done in **[99]** to verify the preservation of a specific notion of system consistency, which we discussed in Section 6. Transactions are generally less expressive than the temporal logics used for model-checking, but typically easier to verify. Here the reconfigurations are specified in Alloy, and the Alloy Analyzer is used to automatically verify that post-conditions are ensured by the reconfiguration if the pre-conditions are initially satisfied, and those pre-conditions are tested at runtime before executing the reconfiguration.

For the highest level of genericity in verification, one must turn to interactive proof assistants. Buisson et al. propose a development process for reconfigurations based on the Coq proof assistant **[32]**. The process is centered around a concrete component model called Pycots, and an abstract component model called Coqcots. Pycots offers Dynamic Software Update (DSU) mechanisms to reconfigure components without completely stopping, as well as introspection features that allow translation towards the abstract model Coqcots. Conversely, the code extraction features of Coq can be used to obtain a Pycots implementation of a system that was previously designed and verified in Coqcots. This bidirectional translation is automated, but the verification process is largely carried out inside Coq. Some correctness properties of the model are proven in the framework, notably the preservation of some structural invariants by the reconfiguration primitives of the model, and the tool offers features to automate some repetitive proofs steps, but application-specific properties must be proven manually, making this technique suitable only for developers that are well-versed in interactive theorem proving.

Instead of verifying a given reconfiguration program, automated techniques can be used to generate programs that satisfy a given specification, yielding correct-by-design reconfigurations. Kikuchi et al. leverage the Alloy Analyzer for this purpose [84], using it to solve a constraint satisfaction problem and extracting a reconfiguration from the solution. No reference model is used, but a very simple component model is assumed, and the user is expected to specify constraints that model components and the semantics of reconfiguration operations. The component model is based on different relationships between components, including "member of" that can encode some hierarchy, but also usage of resources, etc. We call the component model "ad-hoc" because there it has, to the best of our knowledge, no formal definition. It is not clear whether the technique would scale to a more complex component model. A similar synthesis technique has been implemented based on an SMT solver, this time targeting the Concerto component model, and its semantics for inter-component dependencies and parallel execution of reconfigurations operations **[111]**.

Table 5 classifies the aforementioned seven main contributions according to the six criteria introduced in Section 4. Given the emphasis on reconfiguration operations in this area of research,

	Reconf. aspect	*Reconf. op.*	*properties*	*verif. technique*	*expertise*	*comp. model*
[43]	nfunc	connector reconf.	bhv *temporal*	MC (ad hoc algorithm)	FM (ReCTL*)	Reo
[91]	nfunc	connector reconf.	bhv *equivalence*	static analysis (ad hoc algorithm)	none	Reo
[62]	func nfunc	✓	bhv *temporal*	MC (CADP)	FM (temporal logic)	GCM
[6, 29]	func	✓	bhv *pre- and post-conditions*	deduction (Hoare triples)	FM	inspired from BIP and Reo
[99]	func nfunc	✓	bhv *pre- and post-conditions*	RV (transaction manager)	FM (first-order logic/Alloy)	Fractal
[32]	func	✓ + `hotSwap`	bhv *pre- and post-conditions*	ITP (Coq)	FM (Calculus of Constructions)	Coqcots
[49]	lifec	✓ + `pushB` + `wait`	bhv. *temporal*	MC (Romeo)	DSL	Concerto
[111]	lifec	✓ + `pushB` + `wait`	CBD	SMT (Z3)	none	Concerto
[84]	lifec	✓ + 3 ad hoc operations	CBD	SAT, FOL (Alloy)	DSL	ad hoc

✓: yes with the four usual topological operations (add/remove/bind/unbind)

Table 5. Comparison of the contributions to AR3 according to six criteria established in Section 4.

all contributions consider the full range of operations available in the model that they target. The nature of the properties being verified is strongly tied to the verification technique being used, with temporal properties and temporal logics being associated with model-checking, while techniques based on pre- and post-conditions rely on general-purpose logics and associated verification tools. The properties that can be expressed over reconfigurations are extremely rich, and the state-space of possible configurations for an application is very large. Consequently, nearly all techniques require some expertise in formal methods to at least express the appropriate properties, and in some cases to prove them. The exception is **[91]**, which simply checks the equivalence of two reconfigurations, and therefore does not require additional input from the user. A middle ground is found by **[49]**, which offers a DSL to express common and useful temporal properties, thereby enabling non-experts to use model-checking transparently. Efforts in this direction are important since the contributions in this area of research target application developers, who are less likely to have formal methods expertise.

The contributions relying on pre- and post-conditions or deductive verification **[6, 32, 99]** are focused on functional properties although through membranes in the GCM and Fractal models non-functional properties can be considered. By nature, contributions related to Reo **[91]** focus on non-functional properties. In the case of the synthesis of a reconfiguration program, the constraints on the reconfiguration can include non-functional properties **[84]** and in Concerto **[111]**, the execution time of the generated reconfiguration programs is optimised.

9 AR4: INFERENCE OF TARGET CONFIGURATION

Fully automating a reconfiguration requires a mean to determine its target state. Zephyrus **[53]** is a tool developed in the context of the Aeolus project [52] that aims to solve this problem. Zephyrus takes as input (i) a specification of components based on the Aeolus component model [54], (ii) a description of the current state of the system that covers both the infrastructure and component themselves, and (iii) a high-level specification of the targeted state, i.e., a set of constraints and possibly some optimization criteria. From these, Zephyrus generates a Constraint Satisfiability Problem (CSP) using the constraint modeling language MiniZinc. This problem is solved with the constraint solver of GeCode, and Zephyrus uses the solution to output a new correct-by-design target configuration[9].

Kikuchi et al. develop a similar idea **[84]**: from a description of the current state of the system and a set of goal conditions formulated by the different stakeholders of the reconfiguration, a new valid configuration is inferred. Whereas Zephyrus uses a CSP as an intermediate representation, here the constraints are expressed in the Alloy specification language. The Alloy Analyzer is then used to solve these constraints. Unlike Zephyrus, this solution is not limited to inferring a target configuration, but produces at the same time a plan to reach it (see Section 8).

Engage **[59]** is another tool that relies on some form of constraint satisfaction to generate a target configuration. However, the scope of Engage is more limited, as the user is expected to provide a partial target configuration as input, and the aim of the tool is merely to complete it by computing missing dependencies. From the partial configuration, Engage generates a set of boolean constraints given to the SAT solver MiniSat. Engage does not take the current configuration into account, meaning that it is for example not possible to minimize the amount of modifications compared to the current situation. As a result, Engage is quite limited for the purpose of reconfiguration. Additionally, Engage is not based on a recognized component model of the literature, but on an ad hoc hierarchical component-based vision.

The automatic generation of component adapters **[35, 37]** is another approach that generates a part of the configuration, but in this case the generated part is restricted to a communication protocol that handles the behaviorial mismatch of two interfaces that must be connected together. While, from a functional point of view, the generation is limited, existing works are able to handle *at runtime* complex mismatches between the behaviors of two interfaces (changing the order of operations, or merging/splitting requests for example). Some of the works on behavioral adaptation specifically target reconfiguration **[36, 38]**. Considering the nature of the contribution, the authors abstract away component systems as componentized labeled transition systems exporting actions and connected together. While most of the component models can be abstracted this way, their model does not correspond to an existing classified component model. It is worth noticing that the authors expect their approach to be directly applicable to Fractal.

Another approach in this area of research is to develop and use control policies to govern reconfigurations. Ctrl-F **[10]** is a domain specific language to design control policies and build self-adaptive systems. Ctrl-F can be used to specify static aspects of a system as well as dynamic reconfiguration behaviors. For the first part, Ctrl-F relies on a supposedly generic component model that is conceptually close to Fractal. Components as well as a set of valid configurations (assemblies of components) are specified by the developer. The inference of target states is therefore limited to a predetermined set of configurations, an important difference compared to the other solutions described in this section. For the description of reconfiguration behaviors and policies, Ctrl-F includes an event-based language that allows the user to express some rules to pass from one

[9]see online supplementary material for more an illustration of this process

configuration to another[10]. A Ctrl-F specification can be automatically translated to a finite state automata, which may be used for verification and ultimately to synthesize a controller (by using Discrete Controller Synthesis techniques that could be assimilated to model-checking) that will perform the choice of target configurations according to the evolution of the system. The behavior of the system is then controlled so that it stays within the behavior that has been formally defined as safe. The use of Ctrl-F is illustrated by integrating it with Frascati, a reconfiguration framework based on SCA and Fractal component models. Among the approaches we describe in this survey, Ctrl-F is one of the frameworks that particularly focuses on non-functional requirements (see preliminary discussion in Section 1.2). The rule-based approach and the resolution of potential conflicts between rules reflects the difficult trade-offs that exist when addressing non-functional requirements, and proposes an elegant way to handle them.

DR-BIP **[57]** is an extension of BIP with reconfiguration features. It relies on the well defined behavioral semantics of BIP components and adds the concepts of architectural motifs (equivalent to an assembly) on which components are mapped. The reconfiguration operations can modify the set of components involved in the system (add/remove), the architectural motif (by changing connectors), or the mapping of components on the architectural motif. Furthermore, one component can be shared among multiple motifs. In **[58]** a temporal configuration logic (TCL) is introduced to express and verify properties on DR-BIP reconfigurable systems. TCL is a logic with temporal modalities above atomic formulas characterizing the component configurations. TCL formulas can be model-checked at runtime with SMT techniques. DR-BIP performs runtime verification of static adaptation rules, hence ensuring safety in the targeted configuration **[57]**. This approach is similar to Ctrl-F but constantly verifies the configuration of the running system instead of generating a correct-by-design controller.

Dadeau et al. **[50]** also consider adaptive systems and reconfiguration policies. Their approach is quite abstract: configurations and operations can belong to arbitrary sets, even if in the given examples the operations are the usual topological operations. A reconfiguration is a sequence of operations. An adaptation policy rule describes the events and conditions on configurations to satisfy for a reconfiguration to potentially occur with a given priority. This kind of policy is not prescriptive. The contribution of this work is a method to generate tests on a given system, taking into account a usage model, and to return a test verdict with respect to an adaptation policy. The feedback provided makes it possible to detect if reconfigurations occurred even though they were not supposed to, if the priority in a rule relates to the execution frequency of the reconfiguration, and possible inconsistencies in the policy.

Table 6 classifies the aforementioned six main contributions according to five of the six criteria introduced in Section 4. Indeed, the criterion on reconfiguration operations is not applicable to this area of research, as finding the target configuration is decided prior to the set of operations needed to reach this new configuration. All contributions are either based on the idea of correctness-by-design (CBD), by synthesizing either a configuration or a controller, or on the idea of following static adaptation rules that can be verified at runtime. In the case of inferred configurations (CBD), all solution have to tackle the life-cycle aspect of a reconfiguration. Indeed, automatically determining the new configuration to reach is not only driven by the functional and non-functional aspects of a system, but also by information about parametrization of the software, state of the infrastructure, etc. Finally, all these inference-oriented approaches offer a description language that provides an interface with the underlying verification method, and thus correspond to the DSL level of our classification. Importantly, this area of research aims at automating reconfiguration, and the

[10] see online supplementary materials for an illustration of a reconfiguration policy in Ctrl-F

	reconf. aspect	*properties*	*verif. technique*	*expertise*	*comp. model*
[53]	lifec	CBD	CSP (GeCode)	DSL	Aeolus
[84]	lifec	CBD	SAT, FOL (Alloy)	DSL	ad hoc
[59]	lifec	partly CBD	SAT (MiniSAT)	DSL	ad hoc
[35–38]	all	partly CBD	code synthesis	DSL	Connected LTSs
[10]	all	partly CBD	DCS (Heptagon/BZR)	DSL	Fractal-like
[50]	all	bhv.	RV	DSL & FM (temporal logic)	Fractal-like
[57, 58]	func	struct. bhv. *temporal invariants*	SMT, RV (z3, LamaConv)	FM (TCL)	BIP

Table 6. Comparison of the contributions to AR4 according to the criteria established in Section 4. The criterion *reconf. op.* is not applicable to this area of research.

works we present require little user expertise. Users are typically expected to learn a domain-specific language that only exposes them to domain-specific notions related to the objectives of the reconfiguration, and the targeted component model.

10 POSITIONING

The topic of Dynamic Software Update (DSU) was briefly mentioned in this survey, particularly in the presentation of Coqcots [32], which offers a specific reconfiguration operation to apply dynamic substitution of the code of a component, and uses DSU mechanisms to this end. Although DSU is related to reconfiguration, we chose to exclude it from the scope of our classification. Indeed, in Figure 1, DSU amounts to updates in the *modular behavioral modeling* and *modular life-cycle modeling*, outside of the scope of this survey, which is denoted by the purple dashed line. DSU is an important research topic to address dynamic adaptation of distributed software systems and is complementary to contributions studied in this survey.

In the interest of completeness, we briefly mention two contributions related to AR4 that do not conform strictly to the component-oriented scope of this survey: ConfSolve [73] is a declarative language for system configurations (a narrower concept of “configuration” than the one used throughout this survey) with some capabilities for configuration inference, and BtrPlace [72] tackles the problem of virtual machines consolidation with the help of constraint solving. BtrPlace has itself been formally specified and verified by random test generation (based on the formal specification) [71]. More broadly, many papers that explore components or virtual machine placement, as well as task scheduling, are outside the scope of this paper because they do not focus on component models, generic reconfiguration problems, or verification. Most of them exploit mixed linear solvers and heuristics to solve optimization problems.

This study focuses on component-based software engineering. Another related composition models is the concept of *ensembles* [93, 124]. Ensembles provide very dynamic composition of systems where elements of the composition discover each other, and interactions are decided based on many criteria, potentially including physical measures (e.g., spatial distance). Resolution of the composition and component interaction is computed at runtime. In such a model, the notion of configuration is less constrained than in component models and there is no reconfiguration per se because the configuration is constantly evolving dynamically. Consequently, formal techniques for verifying ensembles are quite different from the approaches used in the verification of reconfiguration in component systems.

Finally, feature models [4] are used to encode the possible configurations of features of a software system, by means of mandatory, optional or exclusive features, as well as some constraints between them (e.g., Boolean propositions). A feature model can then be used to extract a satisfaction problem to find one or all valid software configurations, and possibly optimize the choice [4, 61]. In this respect, feature models could be used in the context of AR4 of this survey. However, feature models are not meant to configure the composition of different components but rather the features of a software, although some works have studied the combination and composition of feature models [3]. Compared to feature models, component-based approaches offer a more flexible way of assembling components through their compatible ports. Component models also bring a concrete composition semantics that is not offered by feature models.

11 DISCUSSION AND OPEN CHALLENGES

11.1 Summary

In this survey, we reviewed existing solutions to ensure the safety of reconfigurations of component-based software systems. We organized our survey according to the aspect of reconfiguration targeted by each contribution, namely the formalization of the underlying component model (Preliminary study), the verification of stable configurations (AR1), the kind of application state required to allow safe reconfiguration (AR2), the verification of reconfiguration scripts (AR3), and the safe generation of target configurations (AR4). Through all the studied contributions, we answered the four major questions identified in the introduction:

What assumptions on composition models are made in order to verify reconfigurations? To answer this question, we identified in Section 3 the major component models and their characteristics. We identified the stakeholders that play a role in reconfiguration, and observed how management and reconfiguration can be organized. This presentation of the key concepts must be related both to the formalization of component models, and to their subsequent uses. Section 11.2 below identifies some future research directions related to this question.

What properties on reconfiguration are verified and how? In order to address this question, one first needs to understand which kind of property can be proven by each formal method. We provided the key characteristics of each formal method in Section 2 and identified the formal method used in each summarizing table. We uncovered a great variety of properties verified in the various contributions, and the absence of a standard way to characterize correctness for reconfiguration of component-based systems.

When do verification steps take place, and which expertise is required from stakeholders? To answer this question, it is necessary to understand the role of each stakeholder involved in the development and reconfiguration of a component-based application, which is described in Section 3. We systematically characterized the level of expertise required by the user of the verification techniques we studied. However we did not provide a classification of the complexity of component models themselves, which is an aspect of the global expertise needed to verify a reconfiguration.

11.2 Unified specification and formalization of component models

Most of the existing results are separate initiatives that prove interesting and difficult properties in the context of a dedicated component model and sometimes a dedicated application (or application domain). Some component models like Fractal gather quite a few formalization initiatives, but only a minority of component-based applications are encoded in Fractal. Of course, various component models target different application domains and Fractal could be used to encode some of the applications written in a different model. For example, the Frascati initiative allows users to consider an SCA application as a Fractal component assembly, showing that it is possible to merge the efforts on two existing component models.

However, Fractal is not rich enough to encompass all the achievements presented in this survey, even with a translation from a component model to another. Better standardization of component terminology and structural elements across component models could help, but they are not sufficient. As a simple example, GCM was conceived as an extension of Fractal to include structural aspects that better address applications running on computing grids, and to propose a more structured view of the non-functional aspects of the component assembly, adding support for non-functional reconfigurations to the model.

However, the diversity of results presented here goes beyond these structural aspects. To take into account applications with complex management, one should also be able to reason about complex component life-cycles, and thus a standardization of life-cycle specifications would also be required. Similarly, DevOps management entails various notions that have no equivalent from one component model to another, and are not always a straightforward adaptation of a classical functional aspect. Providing a unified terminology and specification of non-functional aspects and life-cycle aspects seems to be a promising research direction to allow the comparison, and ultimately the composition, of the various techniques presented here.

Agreement on a specification of properties and components would lead to a related question, namely the sharing of formalization results, including tools but also proven results. This concern is very present in the interactive proving community. On the side of specification, Lem [105] is a specification language that offers less expressivity than those of proof assistants, but that is sufficient in its application domain, the semantics of programming languages. Specifications written in Lem can be translated to various provers' specification language, but also compiled to executable specifications in the OCaml functional language. The DeepSpec[11] project focuses on projects within the Coq proof assistant that have rich and complex specifications: the goal is to study how such specifications can be used bidirectionally, i.e., as specifications for client formalizations and also as specifications for an implementation. Finally, Dedukti [120] is a logical framework: different logics can be expressed within Dedukti which allows to glue formalizations and proofs written in various interactive and automated theorem provers.

We believe that one of the first challenges that remains in the formal verification of component models is unification of the different approaches for component model specification. As stated above, multiple formalizations might be warranted because they lend themselves to different properties of interest. However, a unified model where different component models and correctness criteria can be represented in a comparable manner would be extremely valuable.

11.3 Towards completely verified reconfigurable systems

Providing a fully proven framework for reconfigurable components is still an open issue. This would require addressing the problems corresponding to each area of research described in this

[11] https://deepspec.org

survey in one formal framework. A unified view on component systems, as described above, would help reach that goal.

Notably, some works listed in this survey study two overlapping areas of research as we defined them. First, contributions on the verification of GCM by model-checking [11, 12] simultaneously reason about the component model (Preliminary study), prove properties on the behavior of reconfiguration (AR1), but also in the case of [11] verify limited reconfiguration programs, which provides a link between AR1 and AR3 as it verifies the behavioral correctness of a configuration that can perform a simple adaptation step. Similarly, the formalization of Fractal in Alloy [99] allows the authors to reason at the same time on the structure of the component model (Preliminary study) and the structure of a component configuration (AR1 and AR3). Another interesting perspective has been investigated in works that infer target configurations based on specification of an expected behavior. Indeed, both Ctrl-F [10] and DR-BIP [57] mostly address AR4, but they are placed in a setting that requires a reliable formalization of the behavior of each configuration, and thus also partly tackle AR1. To go further, the Fractal model could be used to unify works on Ctrl-F and in Alloy, thus addressing multiple problems at the same time.

Finally, we point out one limitation inherent to the inference of target configurations (AR4) relative to the other areas of research. The problem in AR4 is essentially constraint-solving, but existing solutions do not take into account constraints that result from the safety requirements derived from AR2 (reaching a quiescent state before reconfiguration) and AR3 (performing the reconfiguration in a safe manner), even though they would lead to better choices. However, taking into consideration all these aspects would require modeling both the behavior and the structure of the system, leading to an explosion of the number of intermediate configurations to be considered.

11.4 Verified reconfigurations of component-based applications in real-life systems

Beyond the scientific questions listed in this survey, some practical challenges will have to be addressed to implement real-life systems with verified reconfiguration.

Monitoring. Fully automating reconfiguration in a component-based system requires monitoring both the component system and its environment in order to detect triggering events. Some component models, such as DR-BIP or Fractal, have introspection capabilities that provide crucial information regarding the structure of the component system and some of its internal state. Another approach is to combine a model with existing monitoring solutions. One example is [83], which uses a Configuration Management Database to obtain an accurate description of the system state, and translate it into a first-order logical description, prior to synthesis of the reconfiguration procedure. Furthermore, in [50, 58], both studied in AR4, a verification at runtime is performed on the configuration of the system, which could also be considered as a formal monitoring strategy. However, implementing automated and verified reconfiguration will require more investigation on the integration between system monitoring techniques and verified reconfiguration procedures (AR3).

DevOps stakeholders. When the behavior of components and assemblies is abstracted, typically through a life-cycle modeling, reconfiguration is achieved via a set of basic operations, such as *starting* or *updating*. Such operations are handled on a daily basis by DevOps engineers and systems administrators, either manually, through automated scripts, or with advanced DevOps tools. For this reason, DevOps configuration tools (e.g., Ansible, Puppet) or orchestration tools (e.g., Kubernetes, Nomad) can be compared to the contributions that abstract behavioral aspects through life-cycles [42, 54]. However, automatic DevOps tools are limited in their reconfiguration capabilities for two main reasons: (1) complex interactions between life-cycles can generally not be modeled (although some initiatives in that direction are being studied, e.g., Entrypoints, Argo-CD,

and service mesh in Kubernetes); (2) management operations are highly sensitive, particularly when it comes to critical services, and therefore require high confidence in the automatic reconfiguration tools. Because of the lack of safety in existing solutions, only limited reconfigurations are currently automated, such as scalability operations or virtual machines migrations. *Immutable operations* offer a promising perspective for safety. They perform specific reconfiguration procedures (e.g., upgrade) such that no container, virtual machine, or component is reconfigured in place. Instead, changes in a given element require the deployment of a new component, and are thus safe and easy to roll back. However, this approach is costly and may lead to duplicating a significant part of the system. Combining immutable operations and in place reconfigurations in a safe way could offer a solution to optimize their cost and energy consumption. Safe DevOps procedures can be realized by automatic reconfigurations based on the formal specification of component life-cycles, their interactions with one another, and their operational semantics (as in Concerto [42] for instance). Of course abstracting the behaviors within a life-cycle is challenging, the right abstraction level should be used to limit complexity while allowing the specification of properties of interest to DevOps engineers [49].

Fault tolerance and decentralization. Practical automatic reconfiguration systems need to be tolerant to faults and errors. Immutability and rollback mechanisms are commonly adopted solutions in the DevOps community, but service duplications induce large overheads. Designing safe and light roll back mechanisms for reconfiguration is of high importance, in particular for emerging Edge and IoT infrastructures where resources are limited. In addition to reconfiguration faults, the increasing size of distributed infrastructures and networks will probably lead to more frequent network faults. Facing that challenge requires decentralized solutions to the scientific problems presented in this survey. The only presented work that specifically tackles decentralized reconfiguration is [98], which runs reconfigurations based on local knowledge of the targeted sub-assembly; this work has not been formally verified.

Efficient execution and verification time. Fast reconfigurations are important to quickly reach new targeted states, either for security reasons or simply to reduce disruption of services. This can be achieved by parallelizing reconfiguration operations.

The need for speed of verifying or synthesizing reconfigurations may depend on the context. In the case of a planned reconfiguration for a significant architectural and behavioral evolution of a large scale distributed system, it may be acceptable to spend an important amount of time and resources. In the case of a reconfiguration to patch a defective distributed application, the verification should be fast.

Model-checking even a modest but complete assembly including behavioral aspects requires time and distributed model-checking [11]. Introducing reconfiguration aspects may still be tractable for some properties but not in general [62]. In hierarchical models, it is possible in principle to perform the verification in a compositional way. However, when considering behavioral aspects, it is necessary to abstract the behavior of a composite component to verify its interactions with other components. This abstraction phase is neither automatic nor supported by tools and case studies on very large hierarchical applications are still missing.

One way to scale model-checking is to use parametrized model-checking, to verify properties for an unbound number of components. As this problem is in general undecidable [13], various techniques are designed to handle sub-classes. However using them in practice requires a deep understanding of their abilities and limitations. There exists contributions such as [88] that aim to ease the verification of assemblies with an unbounded number of components but, to our knowledge, reconfigurations are not considered yet.

Interactive theorem proving is another way to handle arbitrary assemblies and reconfigurations. However, its use is often limited to verify specific algorithms (for example an algorithm to synthesize a reconfiguration program) or meta-level properties of reconfiguration languages. Most contributions relying on proof assistants are limited to structural aspects of component applications.

Scalability of the verification or synthesis in the context of the reconfiguration of component-based systems remains a challenge. Leveraging the semantic properties and structure of models as well as studying trade-offs balancing expressivity of the reconfiguration features and ease of verification are promising directions.

REFERENCES

[1] Jean-Raymond Abrial. 1996. *The B-Book.* Cambridge University Press.

[2] Jean-Raymond Abrial. 2010. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press.

[3] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. 2010. Managing Variability in Workflow with Feature Model Composition Operators. In *Software Composition (LNCS)*, Vol. 6144. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–33.

[4] Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. 2013. FAMILIAR: A domain-specific language for large scale management of feature models. *Sci. Comput. Program.* 78, 6 (2013), 657–681. https://doi.org/10.1016/j.scico.2012.12.004

[5] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. 2016. *Deductive Software Verification – The KeY Book.* LNCS, Vol. 10001. Springer.

[6] Emma Ahrens, Marius Bozga, Radu Iosif, and Joost-Pieter Katoen. 2021. Local reasoning about parameterized reconfigurable distributed systems. *CoRR* (2021).

[7] Farah Ait Salaht, Frédéric Desprez, and Adrien Lebre. 2020. An overview of service placement problem in Fog and Edge Computing. *ACM Computing Surveys* 53, 3, Article 65 (June 2020). https://doi.org/10.1145/3391196

[8] Hamzeh Alabool, Ahmad Kamil, Noreen Arshad, and Deemah Alarabiat. 2018. Cloud service evaluation method-based Multi-Criteria Decision-Making: A systematic literature review. *Journal of Systems and Software* 139 (2018), 161–188. https://doi.org/10.1016/j.jss.2018.01.038

[9] Rajeev Alur, Costas Courcoubetis, and David Dill. 1990. Model-checking for real-time systems. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science (LICS '90).* IEEE, 414–425. https://doi.org/10.1109/LICS.1990.113766

[10] Frederico Alvares, Eric Rutten, and Lionel Seinturier. 2017. A Domain-specific Language for The Control of Self-adaptive Component-based Architecture. *Journal of Systems and Software* 130 (Jan. 2017), 94–112. https://doi.org/10.1016/j.jss.2017.01.030

[11] Rabéa Ameur-Boulifa, Raluca Halalai, Ludovic Henrio, and Eric Madelaine. 2011. Verifying Safety of Fault-Tolerant Distributed Components. In *International Symposium on Formal Aspects of Component Software (FACS 2011) (LNCS).* Springer, Berlin, Heidelberg, 278–295. https://doi.org/10.1007/978-3-642-35743-5_17

[12] Rabea Ameur-Boulifa, Ludovic Henrio, Oleksandra Kulankhina, Eric Madelaine, and A. Savu. 2017. Behavioural semantics for asynchronous components. *Journal of Logical and Algebraic Methods in Programming* 89 (2017), 1–40. https://doi.org/10.1016/j.jlamp.2017.02.003

[13] Krzysztof R. Apt and Dexter C. Kozen. 1986. Limits for automatic verification of finite-state concurrent systems. *Inform. Process. Lett.* 22, 6 (1986), 307–309. https://doi.org/10.1016/0020-0190(86)90071-2

[14] Farhad Arbab. 2004. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science* 14, 3 (2004), 329–366. https://doi.org/10.1017/S0960129504004153

[15] Farhad Arbab, Roberto Bruni, Dave Clarke, Ivan Lanese, and Ugo Montanari. 2008. Tiles for Reo. In *International Workshop on Algebraic Development Techniques (WADT) (LNCS)*, Vol. 5389. Springer, Berlin, Heidelberg, 37–55. https://doi.org/10.1007/978-3-642-03429-9_4

[16] Farhad Arbab and Jan JMM Rutten. 2002. A Coinductive Calculus of Component Connectors. In *International Workshop on Algebraic Development Techniques (WADT) (LNCS)*, Vol. 2755. Springer, Berlin, Heidelberg, 34–55. https://doi.org/10.1007/978-3-540-40020-2_2

[17] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Ckecking.* MIT Press.

[18] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. 2006. Modeling component connectors in Reo by constraint automata. *Sci. Comput. Program.* 61, 2 (2006), 75–113. https://doi.org/10.1016/j.scico.2005.10.008

[19] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. 2018. Introduction to Runtime Verification. In *Lectures on Runtime Verification.* LNCS, Vol. 10457. Springer, Cham, 1–33. https://doi.org/10.1007/978-3-319-75632-5_1

[20] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. 2011. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Softw.* 28, 3 (May 2011), 41–48. https://doi.org/10.1109/MS.2011.27

[21] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. 2009. GCM: a grid extension to Fractal for autonomous distributed components. *Annals of telecommunications* 64, 1-2 (2009), 5–24. https://doi.org/10.1007/s12243-008-0068-8

[22] Sebastian S. Bauer, Rolf Hennicker, and Stephan Janisch. 2010. Behaviour Protocols for Interacting Stateful Components. *Electron. Notes Theor. Comput. Sci.* 263 (2010), 47–66. https://doi.org/10.1016/j.entcs.2010.05.004

[23] David E. Bernholdt, Benjamin A. Allan, Robert Armstrong, Felipe Bertrand, Kenneth Chiu, Tamara L. Dahlgren, Kostadin Damevski, Wael R. Elwasif, Thomas G. W. Epperly, Madhusudhan Govindaraju, Daniel S. Katz, James A. Kohl, Manoj Krishnan, Gary Kumfert, J. Walter Larson, Sophia Lefantzi, Michael J. Lewis, Allen D. Malony, Lois C. Mclnnes, Jarek Nieplocha, Boyana Norris, Steven G. Parker, Jaideep Ray, Sameer Shende, Theresa L. Windus, and Shujia Zhou. 2006. A Component Architecture for High-Performance Scientific Computing. *Intl J. of High Performance Computing Applications* 20, 2 (2006), 163–202. https://doi.org/10.1177/1094342006064488

[24] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development.* Springer. https://doi.org/10.1007/978-3-662-07964-5

[25] Julien Bigot, Zhengxiong Hou, Christian Pérez, and Vincent Pichon. 2014. A low level component model easing performance portability of HPC applications. *Computing* 96, 12 (2014), 1115–1130. https://doi.org/10.1007/s00607-013-0368-3

[26] Simon Bliudze, Anastasia Mavridou, Radoslaw Szymanek, and Alina Zolotukhina. 2017. Exogenous coordination of concurrent software components with JavaBIP. *Software: Practice and Experience* 47, 11 (2017), 1801–1836. https://doi.org/10.1002/spe.2495

[27] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. 2014. Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer (STTT)* (2014). https://doi.org/10.1007/s10009-014-0int314-5

[28] Fabienne Boyer, Olivier Gruber, and Damien Pous. 2013. Robust reconfigurations of component assemblies. In *35th International Conference on Software Engineering (ICSE).* IEEE Computer Society, 13–22. https://doi.org/10.1109/ICSE.2013.6606547

[29] Marius Bozga, Lucas Bueri, and Radu Iosif. 2022. Decision Problems in a Logic for Reasoning about Reconfigurable Distributed Systems. In *International Joint Conference on Automated Reasoning*. Springer, 691–711.

[30] Sally C Brailsford, Chris N Potts, and Barbara M Smith. 1999. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research* 119, 3 (1999), 557–581. https://doi.org/10.1016/S0377-2217(98)00364-6

[31] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. 2006. The FRACTAL component model and its support in Java. *Software: Practice and Experience* 36, 11-12 (2006), 1257–1284. https://doi.org/10.1002/spe.767

[32] Jérémy Buisson, Fabien Dagnat, Elena Leroux, and Sébastien Martinez. 2015. Safe reconfiguration of Coqcots and Pycots components. *Journal of Systems and Software* 122 (2015), 430–444. https://doi.org/10.1016/j.jss.2015.11.039

[33] G. A. Bundell, G. Lee, J. Morris, K. Parker, and Peng Lam. 2000. A software component verification tool. In *Proceedings International Conference on Software Methods and Tools (SMT 2000).* IEEE, 137–146. https://doi.org/10.1109/SWMT.2000.890429

[34] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. 2006. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Fourth International Conference on Software Engineering Research, Management and Applications (SERA '06).* IEEE, 40–48. https://doi.org/10.1109/SERA.2006.62

[35] Javier Cámara, Gwen Salaün, and Carlos Canal. 2008. Composition and Run-time Adaptation of Mismatching Behavioural Interfaces. *J. Univers. Comput. Sci.* 14, 13 (2008), 2182–2211. https://doi.org/10.3217/jucs-014-13-2182

[36] Carlos Canal, Javier Cámara, and Gwen Salaün. 2012. Structural reconfiguration of systems under behavioral adaptation. *Sci. Comput. Program.* 78, 1 (2012), 46–64. https://doi.org/10.1016/j.scico.2011.09.003

[37] Carlos Canal, Pascal Poizat, and Gwen Salaün. 2006. Synchronizing Behavioural Mismatch in Software Composition. In *8th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS) (LNCS)*, Vol. 4037. Springer, Berlin, Heidelberg, 63–77. https://doi.org/10.1007/11768869_7

[38] Antonio Cansado, Carlos Canal, Gwen Salaün, and Javier Cubo. 2010. A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation. *Electron. Notes Theor. Comput. Sci.* 263 (2010), 95–110. https://doi.org/10.1016/j.entcs.2010.05.006

[39] Sagar Chaki, Edmund M Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2004. Modular verification of software components in C. *IEEE Transactions on Software Engineering* 30, 6 (2004), 388–402. https://doi.org/10.1109/ICSE.2003.1201217

[40] Patrice Chalin and Frédéric Rioux. 2008. JML Runtime Assertion Checking: Improved Error Reporting and Efficiency Using Strong Validity. In *FM 2008: Formal Methods (LNCS)*, Vol. 5014. Springer, Berlin, Heidelberg, 246–261. https://doi.org/10.1007/978-3-540-68237-0_18

[41] Maverick Chardet, Hélène Coullon, and Christian Pérez. 2020. Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto. In *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid 2020)*. IEEE, Melbourne, Australia. https://doi.org/10.1109/CCGrid49817.2020.00-59

[42] Maverick Chardet, Hélène Coullon, and Simon Robillard. 2021. Toward safe and efficient reconfiguration with Concerto. *Sci. Comput. Program.* 203 (2021), 102582. https://doi.org/10.1016/j.scico.2020.102582

[43] Dave Clarke. 2008. A Basic Logic for Reasoning About Connector Reconfiguration. *Fundamenta Informaticae* 82, 4 (2008), 361–390.

[44] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model Checking* (2nd ed.). MIT Press.

[45] Lori A. Clarke and David S. Rosenblum. 2006. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes* 31, 3 (2006), 25–37. https://doi.org/10.1145/1127878.1127900

[46] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs) (LNCS)*, Vol. 5674. Springer, Berlin, Heidelberg, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

[47] Loïc Correnson and Julien Signoles. 2012. Combining Analyses for C Program Verification. In *Formal Methods for Industrial Critical Systems (FMCIS) (LNCS)*, Vol. 7437. Springer, Berlin, Heidelberg, 108–130. https://doi.org/10.1007/978-3-642-32469-7_8

[48] Hélène Coullon, Julien Bigot, and Christian Perez. 2019. Extensibility and Composability of a Multi-Stencil Domain Specific Framework. *Intl J. of Parallel Programming* 47 (2019), 1046–1085. https://doi.org/10.1007/s10766-017-0539-5

[49] Hélène Coullon, Claude Jard, and Didier Lime. 2019. Integrated Model-checking for the Design of Safe and Efficient Distributed Software Commissioning. In *15th International Conference on integrated Formal Methods (IFM) (LNCS)*, Vol. 11918. Springer, Bergen, Norway, 120–137. https://doi.org/10.1007/978-3-030-34968-4_7

[50] Frédéric Dadeau, Jean-Philippe Gros, and Olga Kouchnarenko. 2020. Testing adaptation policies for software components. *Software Quality Journal* 28, 3 (2020), 1347–1378. https://doi.org/10.1007/s11219-019-09487-w

[51] George Bernard Dantzig. 1998. *Linear programming and extensions*. Vol. 48. Princeton university press.

[52] Roberto Di Cosmo, Antoine Eiche, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro, and Jakub Zwolakowski. 2015. Automatic Deployment of Services in the Cloud with Aeolus Blender. In *Service-Oriented Computing (ICSOC) (LNCS)*, Vol. 9435. Springer, Berlin, Heidelberg, 397–411. https://doi.org/10.1007/978-3-662-48616-0_28

[53] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. 2014. Automated Synthesis and Deployment of Cloud Applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 211–222. https://doi.org/10.1145/2642937.2642980

[54] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. 2014. Aeolus: a Component Model for the Cloud. *Information and Computation* 239 (1 2014), 100–121. https://doi.org/10.1016/j.ic.2014.11.002

[55] Zuohua Ding, Zhenbang Chen, and Jing Liu. 2008. A Rigorous Model of Service Component Architecture. *Electron. Notes Theor. Comput. Sci.* 207 (2008), 33–48.

[56] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

[57] Rim El Ballouli, Saddek Bensalem, Marius Bozga, and Joseph Sifakis. 2018. Programming Dynamic Reconfigurable Systems. In *Formal Aspects of Component Software - 15th International Conference (FACS) (LNCS)*, Vol. 11222. Springer, Cham, 118–136. https://doi.org/10.1007/978-3-030-02146-7_6

[58] Antoine El-Hokayem, Marius Bozga, and Joseph Sifakis. 2021. A Temporal Configuration Logic for Dynamic Reconfigurable Systems. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21)*. ACM, New York, NY, USA, 1419–1428. https://doi.org/10.1145/3412841.3442017

[59] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. 2012. Engage: a deployment management system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/2254064.2254096

[60] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. 2008. Deploying on the Grid with DeployWare. In *Proceedings of the Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID '08)*. IEEE Computer Society, Washington, DC, USA, 177–184. https://doi.org/10.1109/CCGRID.2008.59

[61] José A. Galindo, Mathieu Acher, Juan Manuel Tirado, Cristian Vidal, Benoit Baudry, and David Benavides. 2016. Exploiting the Enumeration of All Feature Model Configurations: A New Perspective with Distributed Computing. In *Proceedings of the 20th International Systems and Software Product Line Conference (SPLC '16)*. ACM, New York, NY, USA, 74–78. https://doi.org/10.1145/2934466.2934478

[62] Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. 2013. Formally Reasoning on a Reconfigurable Component-Based System – A Case Study for the Industrial World. In *Formal Aspects of Component Software (FACS) (LNCS)*, Vol. 8348. Springer, Cham, 137–156. https://doi.org/10.1007/978-3-319-07602-7_10

[63] Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. 2014. Bringing Coq into the World of GCM Distributed Applications. *Intl J. of Parallel Programming* 42, 4 (2014), 643–662. https://doi.org/10.1007/s10766-013-0264-7

[64] Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. 2015. Painless Support for Static and Runtime Verification of Component-Based Applications. In *Fundamentals of Software Engineering (FSEN) (LNCS)*, Vol. 9392. Springer, Cham, 259–274. https://doi.org/10.1007/978-3-319-24644-4_18

[65] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing (SoCC '16)*. ACM, New York, NJ, USA, 1–16. https://doi.org/10.1145/2987550.2987583

[66] Sandeep Gupta. 2022. Non-functional requirements elicitation for edge computing. *Internet of Things* 18 (2022), 100503. https://doi.org/10.1016/j.iot.2022.100503

[67] Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. 2013. HATS Abstract Behavioral Specification: The Architectural View. In *Formal Methods for Components and Objects (FMCO 2011) (LNCS)*, Vol. 7542. Springer, Berlin, Heidelberg, 109–132. https://doi.org/10.1007/978-3-642-35887-6_6

[68] Ludovic Henrio, Florian Kammüller, and Muhammad Uzair Khan. 2010. A Framework for Reasoning on Component Composition. In *Formal Methods for Components and Objects (FMCO 2009) (LNCS)*, Vol. 6286. Springer, Berlin, Heidelberg, 1–20. https://doi.org/10.1007/978-3-642-17071-3_1

[69] Ludovic Henrio and Marcela Rivera. 2008. Stopping safely hierarchical distributed components: application to GCM. In *CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*. ACM, New York, NY, USA. https://doi.org/10.1145/1456190.1456201

[70] Thomas A Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. 1994. *Symbolic model checking for real-time systems*. Technical Report. Cornell University.

[71] Fabien Hermenier and Ludovic Henrio. 2017. Trustable Virtual Machine Scheduling in a Cloud. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM, New York, NJ, USA, 15–26. https://doi.org/10.1145/3127479.3128608

[72] Fabien Hermenier, Julia Lawall, and Gilles Muller. 2013. BtrPlace: A Flexible Consolidation Manager for Highly Available Applications. *IEEE Transactions on Dependable and Secure Computing* 10, 5 (2013), 273–286. https://doi.org/10.1109/TDSC.2013.5

[73] John A. Hewson, Paul Anderson, and Andy Gordon. 2012. A Declarative Approach to Automated Configuration. In *Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques* (proceedings of the 26th international conference on large installation system administration (lisa): strategies, tools, and techniques ed.) *(LISA '12)*. USENIX Association, USA, 51–66. https://www.usenix.org/conference/lisa12/technical-sessions/presentation/hewson

[74] Gerald H Hilderink. 2006. Software Specification Refinement and Verification Method with I-Mathic Studio.. In *CPA*. 297–310.

[75] Matias Ibanez, Cristian Ruz, Ludovic Henrio, and Javier Bustos-Jiménez. [n. d.]. ([n. d.]).

[76] Daniel Jackson. 2012. *Software Abstractions* (revised ed.). MIT Press.

[77] Pavel Ježek, Jan Kofroň, and František Plášil. 2006. Model Checking of Component Behavior Specification: A Real Life Experience. *Electron. Notes Theor. Comput. Sci.* (2006), 197–210. https://doi.org/10.1016/j.entcs.2006.05.023

[78] Kenneth Johnson and Radu Calinescu. 2014. Efficient Re-resolution of SMT Specifications for Evolving Software Architectures. In *Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures (QoSA '14)*. ACM, New York, NJ, USA, 93–102. https://doi.org/10.1145/2602576.2602578

[79] Kenneth Johnson, Radu Calinescu, and Shinji Kikuchi. 2013. An Incremental Verification Framework for Component-Based Software Systems. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE '13)*. ACM, New York, NJ, USA, 33–42. https://doi.org/10.1145/2465449.2465456

[80] Sung-Shik TQ Jongmans and Farhad Arbab. 2012. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science* 22, 1 (2012). https://doi.org/10.7561/SACS.2012.1.201

[81] Joost-Pieter Katoen. 2016. The probabilistic model checking landscape. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. ACM, New York, NJ, USA, 31–45. https://doi.org/10.1145/2933575.2934574

[82] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019). https://doi.org/0.1017/S0956796818000229

[83] Shinji Kikuchi and Kunihiko Hiraishi. 2014. Improving Reliability in Management of Cloud Computing Infrastructure by Formal Methods. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 1–7. https://doi.org/10.1109/NOMS.2014.6838285

[84] Shinji Kikuchi, Satoshi Tsuchiya, and Kunihiko Hiraishi. 2013. Synthesis of Configuration Change Procedure Using Model Finder. *IEICE Transactions on Information and Systems* 96, 8 (2013), 1696–1706. https://doi.org/10.1587/transinf.E96.D.1696

[85] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal Asp. Comput.* 27, 3 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

[86] Christian Koehler, Farhad Arbab, and Erik de Vink. 2008. Reconfiguring Distributed Reo Connectors. In *International Workshop on Algebraic Development Techniques (WADT) (LNCS)*, Vol. 5486. Springer, Berlin, Heidelberg, 221–235. https://doi.org/Pisa,Italy

[87] Natallia Kokash, Christian Krause, and Erik De Vink. 2012. Reo+ mCRL2: A Framework for Model-Checking Dataflow in Service Compositions. *Formal Aspects of Computing* 24, 2 (2012), 187–216. https://doi.org/0.1007/s00165-011-0191-6

[88] Igor Konnov, Tomer Kotek, Qiang Wang, Helmut Veith, Simon Bliudze, and Joseph Sifakis. 2016. Parameterized Systems in BIP: Design and Model Checking. In *27th International Conference on Concurrency Theory (CONCUR 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 59. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 30:1–30:16. https://doi.org/10.4230/LIPIcs.CONCUR.2016.30

[89] Jeff Kramer and Jeff Magee. 1990. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering* 16, 11 (1990), 1293–1306.

[90] Nane Kratzke and Peter-Christian Quint. 2017. Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. *Journal of Systems and Software* 126 (2017), 1–16. https://doi.org/10.1016/j.jss.2017.01.001

[91] Christian Krause, Holger Giese, and Erik De Vink. 2013. Compositional and behavior-preserving reconfiguration of component connectors in Reo. *Journal of Visual Languages & Computing* 24, 3 (2013), 153–168. https://doi.org/0.1016/j.jvlc.2012.09.002

[92] Christian Krause, Ziyan Maraikar, Alexander Lazovik, and Farhad Arbab. 2011. Modeling Dynamic Reconfigurations in Reo Using High-Level Replacement Systems. *Sci. Comput. Program.* 76, 1 (2011), 23–36. https://doi.org/10.1016/j.scico.2009.10.006

[93] Filip Krijt, Zbynek Jiracek, Tomas Bures, Petr Hnetynka, and Frantisek Plasil. 2017. Automated Dynamic Formation of Component Ensembles. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*. SciTePress, Setubal, Portugal, 561–568. https://doi.org/10.5220/0006273705610568

[94] Oleksandra Kulankhina. 2016. *A framework for rigorous development of distributed components : formalisation and tools*. Ph.D. Dissertation. Université Côte d'Azur. https://tel.archives-ouvertes.fr/tel-01419298

[95] Vipin Kumar. 1992. Algorithms for Constraint-Satisfaction Problems: A Survey. *AI Magazine* 13, 1 (April 1992), 32–44.

[96] Arnaud Lanoix and Olga Kouchnarenko. 2014. Component Substitution through Dynamic Reconfigurations. In *Proceedings 11th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA) (EPTCS)*, Vol. 147. 32–46. https://doi.org/10.4204/EPTCS.147.3

[97] Vincent Lanore and Christian Pérez. 2015. *A Calculus Enabling Reuse and Composition of Component Assembly Specialization Processes*. Research Report RR-8761. Inria. 21 pages. https://hal.inria.fr/hal-01179483

[98] Vincent Lanore and Christian Pérez. 2015. A Reconfigurable Component Model for HPC. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE '15)*. ACM, New York, NJ, USA, 1–10. https://doi.org/10.1145/2737166.2737169

[99] Marc Léger, Thomas Ledoux, and Thierry Coupaye. 2010. Reliable Dynamic Reconfigurations in a Reflective Component Model. In *Component-Based Software Engineering (CBSE) (LNCS)*, Vol. 6092. Springer, Berlin, Heidelberg, 74–92. https://doi.org/10.1007/978-3-642-13238-4_5

[100] K. Rustan M. Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) (LNCS)*, Vol. 6355. Springer, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[101] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[102] Pierre Letouzey. 2008. Extraction in Coq: an Overview. In *Conference on Computability in Europe (LNCS)*, Vol. 5028. Springer, Berlin, Heidelberg, 359–369. https://doi.org/978-3-540-69407-6_39

[103] Xiaoxing Ma, Luciano Baresi, Carlo Ghezzi, Valerio Panzica La Manna, and Jian Lu. 2011. Version-consistent dynamic reconfiguration of component-based distributed systems. In *19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 245–255. https://doi.org/10.1145/2025113.2025148

[104] Philippe Merle and Jean-Bernard Stefani. 2008. *A formal specification of the Fractal component model in Alloy*. Research Report RR-6721. INRIA. https://hal.inria.fr/inria-00338987

[105] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-World Semantics. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 175–188. https://doi.org/10.1145/2628136.2628143

[106] Tobias Nipkow, Lawrence C. Paulson, and Makarius Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45949-9

[107] OASIS. 2015. Service Component Architecture (SCA). http://www.oasis-opencsa.org/sca

[108] Object Management Group. 2006. CORBA Component Model Specification. http://www.omg.org/spec/CCM/4.0/PDF

[109] Object Management Group. 2006. Deployment and Configuration of component-based Distributed Applications. https://www.omg.org/spec/DEPL/4.0/PDF

[110] Xavier Rival and Kwangkeun Yi. 2020. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press.

[111] Simon Robillard and Hélène Coullon. 2022. SMT-Based Planning Synthesis for Distributed System Reconfigurations. In *Fundamental Approaches to Software Engineering (FASE)*. Springer, 268–287. https://doi.org/10.1007/978-3-030-99429-7_15

[112] Francesca Rossi, Peter Van Beek, and Toby Walsh. 2006. *Handbook of constraint programming*. Elsevier.

[113] Ondrej Rysavy and Jaroslav Rab. 2008. A component-based approach to verification of embedded control systems using TLA⁺. In *2008 International Multiconference on Computer Science and Information Technology*. 719–725. https://doi.org/10.1109/IMCSIT.2008.4747321

[114] Farah Aït Salaht, Frédéric Desprez, and Adrien Lebre. 2020. An Overview of Service Placement Problem in Fog and Edge Computing. *ACM Comput. Surv.* 53, 3, Article 65 (jun 2020), 35 pages. https://doi.org/10.1145/3391196

[115] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. 2012. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience* 42, 5 (2012), 559–583. https://doi.org/10.1002/spe.1077

[116] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. 2017. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs (tool paper). In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (Kalpa Publications in Computing)*, Vol. 3. EasyChair, 164–173. https://doi.org/10.29007/fpdh

[117] Marc Snir, Steve Otto, Huss-Lederman, David Walker, and Jack Dongarra. 1998. *MPI the Complete Reference – Volume 1, The MPI Core* (2nd ed.). MIT Press.

[118] Clemens Szyperski. 2002. *Component Software: Beyond Object-Oriented Programming* (2nd ed.). Addison-Wesley Longman Pub. Co., Inc., Boston, MA, USA.

[119] The Coq Development Team. [n. d.]. The Coq Proof Assistant. http://coq.inria.fr.

[120] François Thiré. 2020. *Interoperability between proof systems using the logical framework Dedukti*. Ph.D. Dissertation. ENS Paris-Saclay. https://hal.archives-ouvertes.fr/tel-03224039

[121] Antonio Vallecillo, Vasco T Vasconcelos, and António Ravara. 2006. Typing the behavior of software components using session types. *Fundamenta Informaticæ* 73, 4 (2006), 583–598.

[122] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. 2007. Tranquillity: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering* 33, 12 (2007), 856–868.

[123] Abderrahim Ait Wakrime, Sébastien Limet, and Sophie Robert. 2015. On the fly reconfiguration of interactive scientific visualization applications. In *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 468–475. https://doi.org/10.1109/HPCSim.2015.7237078

[124] Martin Wirsing, Matthias Hölzl, Mirco Tribastone, and Franco Zambonelli. 2013. ASCENS: Engineering Autonomic Service-Component Ensembles. In *Formal Methods for Components and Objects (FMCO 2011) (LNCS)*, Vol. 7542. Springer, Berlin, Heidelberg, 1–24. https://doi.org/10.1007/978-3-642-35887-6_1

[125] Derek Wyatt. 2013. *Akka Concurrency*. Artima.

[126] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 283–294. https://doi.org/10.1145/1993498.1993532