Ensuring Consistency of Dynamic Reconfiguration of Component-Based Systems

Hamza Zerguine
MOVEP laboratory
USTHB University
Algiers, Algeria,
Email: hzerguine@usthb.dz

Nabila Salmi
MOVEP Laboratory, USTHB
Algiers, Algeria,
LISTIC Laboratory, Université de Savoie
Annecy le Vieux, France
Email: nsalmi@usthb.dz

Malika Ioualalen
MOVEP laboratory
USTHB University
Algiers, Algeria
Email: mioualalen@usthb.dz

***Abstract*—The introduction of dynamic reconfiguration properties in a system can affect its performance and quality of service offered to users. Thus, performance prediction of component-based systems after reconfiguration is important to help software engineers to analyze their applications at the moment of reconfiguration and take decision to keep or discard the analyzed reconfiguration, so that performance problems are avoided. In this case, the design and verification of functional and non-functional properties before and after reconfiguration become a challenge. In particular, when a applying a reconfiguration on a system, the consistency of the new resulting architecture should be checked. To this aim, we describe, in this paper, a generic reconfiguration analysis approach which allows to check the reconfiguration consistency of a component-based architecture, starting from the architectural description of a component-based system. A case study of a system reconfiguration illustrates the effectiveness of our approach.**

***Keywords*-Component-Based Systems; dynamic reconfiguration; formalization; consistency.**

I. INTRODUCTION

Component-based approaches [1] are more and more essential for the development of systems and applications, to meet the challenges of engineering systems such as administration, autonomy. In this paradigm, components are developed in isolation or reused and are then assembled to build a *Component Based System* (CBS). Their objective is to enable a high degree of reusability of the software, rapid development (reducing the cost in terms of development time) and high quality since development is based on precompiled components In this direction, numerous component models have been proposed (e.g., Enterprise Java Beans (EJB) [2], Corba Component Model (CCM) [3], Fractal [4], etc.). They operate different life-cycle stages, target different technical domains (embedded systems, distributed systems, etc.) and offer different degrees of tool support (textual modeling, graphical modeling, automated performance simulation, etc.).

Nowadays, systems need more and more to adapt their behaviour to their environment changes. To do that, they should dynamically add, remove or recompose components by the use of computational reflection. These abilities are called dynamic or runtime reconfiguration and constitute a key element to enable the adaptation of complex systems, such as embedded systems (mobile phones, PDAs, etc.) and service-oriented systems, to a changing environment. Moreover, dynamic system reconfiguration allows to achieve continuous availability of systems.

Dynamic reconfiguration techniques are promising solutions for building highly adaptable component-based systems. However, the introduction of dynamic reconfiguration properties in a system can affect its performance and quality of service offered to users. To avoid this, the design and verification of functional and non-functional properties of a reconfigured system become a challenge.

In this context, our long-term goal is to develop a methodology which allows analysis of component-based applications and their correction after reconfiguration, to help the decision to keep or discard the analyzed reconfiguration. The first property to ensure during analysis of such systems is consistency, which is defined as remaining compliant with their specification [5]. In this paper, we introduce a new formalism for checking consistency of dynamic reconfigurations of component-based systems. We provide this formalism for general component systems characterized by the most common component properties.

Outline. The structure of the paper is as follows. We discuss in Section II the related work. Then, we present in Section III the most important concepts of component-based systems. We detail our approach in Section IV and illustrate it in Section V. We conclude in Section VI and give future works.

II. RELATED WORK

Several approaches were proposed, during last years, for analysis of CBS; a few of them addressed dynamic reconfiguration.

In this context, two main proposals were given for dynamic reconfiguration analysis of CBS. First, Grassi et al. [6] proposed a metamodel called KLAPER, which includes a kernel modelling language. The main goal of this language is to act as a bridge between design models of component-based systems (built using heterogenous languages like Unified Modeling Language (UML) [7], Ontology Web Language (OWL) (OWL-S) [8], etc.) and performance analysis models (Markov chains [9], queueing networks [10], etc.). This first work did not address reconfiguration cases study. Later, in [11], an extension of KLAPER, called D-KLAPER, was given to support the model-based analysis of reconfigurable component-based systems, with a focus on the assessment of particular non-functional properties, namely performance and reliability.

 ISBN: 978-1-61208-304-9

The second work, defined by Leger [5], targeted dynamic reconfigurations reliability analysis for component-based systems, where an analysis approach for the Fractal component model was defined. The approach is summarized in three steps: the first step is a Fractal configuration modeling [5] step of a component-based configuration architecture; then, definition of mechanisms used for maintaining systems consistency during dynamic reconfigurations; finally, implementation of these mechanisms for checking reliable reconfiguration.

Besides, some other approaches were proposed for CBS for checking, in particular, consistency of CBS during dynamic reconfiguration. Warren et al. [12] proposed to do automatic runtime checks of reconfigurable component-based systems for the OpenRec framework [13]. A formal model based on ALLOY [14] was defined for that purpose. It allows architecture constraints expression and checking. Another work [15] has introduced an extension of the Fractal model [16], called *Safran* to enable the development of adaptive applications. It consists of a dedicated programming language for adaptation policies, as well as a mechanism to dynamically attach or detach policies to or from Fractal basic components. Finally, M. Simonot et al. [17] proposed a formal framework, called *Fracl*, for specifying and reasoning about dynamic reconfiguration programs, being written in a Fractal-like programming style [4]. This framework is based on a first order logic, and allows properties specification and proof concerning either functional or control concerns. An encoding of their model using the Focal specification framework [18] enabled them to prove its coherence and obtain a framework for reasoning on concrete architectures.

These proposals are interesting, however, Safran, Fracl and Leger's proposals are focused on Fractal models only. In particular, Fracl was defined only for applications with primitive components. In addition, no difference is done between *Mandatory* and *Optional* interfaces and no subtyping notion is considered. Warren et al. [12] focused on OpenRec framework only. Moreover, only connections between component are modelled and not component behaviours.

In our case, we target to provide a generic formalism to be used for checking consistency in any component-based system. Our approach formalizes main component elements (component, interfaces, bindings, etc.) and defines for each reconfiguration operation a set of constraints to build consistent configurations. Global constraints are also introduced on a CBS after its reconfiguration.

III. Component Based Systems

A software *component* is defined as a unit of composition, provided with contractually specified *interfaces* and explicit context *dependencies* [19]. An interface is an access point to the component, which defines provided or required services. In addition, types, constraints and semantics are defined by the *component model* in order to describe the expected behaviour at runtime.

Interfaces of a component allow to connect it to other components. Consequently, we build a Component-based System by connecting the interfaces of components. These connections are done depending on interactions between components. Generally, two main styles of interactions are defined in component models: synchronous interactions provided by service invocation (such as an Remote Procedure Call (RPC) or Remote Method invocation (RMI) communication), and asynchronous interactions given through notification of events (asynchronous messages). Service invocations take place between a *client* interface requesting a service and a *server* interface providing the service. Besides, event communications are defined between one or more *event source* interfaces generating events and one or several *event sink* interfaces receiving event notifications. The reception of a notification causes the acknowledgment of the reception and execution of a specified reaction called the *handler* of the event. Some event services can use *event channels* for mediating event messages between sources and sinks. An event channel is an entity responsible for registering subscriptions of a specific type of event, receiving events, filtering events according to specific modes, and routing them to the interested sinks.

A component can contain itself a finite number of other interacting components, called *sub-components*, allowing the components to be nested at an arbitrary level. In this case, it is said a *composite* component. At the lowest level, components are said *primitive*. Sometimes, assembling two components may require an adaptation of associated interfaces, whenever these interfaces cannot directly communicate for example. In this case, the adaptation is done with an extra entity, called *connector*, modelling the interaction protocol between the two components.

For each component model, a corresponding *Architecture Description Language (ADL)* allows to describe an assembly of components forming an application. From such a description, a set of tools are used to compile and generate the application code, while checking syntactical and even some semantical properties.

IV. Formalization

Our goal is to propose a new formalism for checking consistency of dynamic reconfigurations of component-based. For this purpose, we give first a set of concepts and then define our approach for checking consistency of CBS.

A. Concepts

1) Component-based configuration:

Definition 1. *A component-based configuration of a system S is defined as a triplet:*

$$Cg = \prec C, I, B \succ \quad where$$

- *C: is a set of components;*
- *I: is a set of interfaces;*
- *B: is a set of component connections or bindings.*

Definition 2. *A component c is defined as:*

$$c = \prec name, granul, state \succ$$

where:

- *name: is the unique name of the component C;*
- *granul: refers to granularity wich can be Composite or Primitive;*
- *state: is the current state of the component C, which can be Started or Stopped.*

Definition 3. *A component interface i is defined as:*

$$i = \prec itfc, role, visib, card, contig, sign \succ$$

where:

- *itfc: is the unique identifier of the interface (being of the form: component-name.interface-name);*
- *role: can be Client / Server (in the case of a service invocation interface) or Sink / Source (in the case of an event based interface);*
- *visib: refers to the visibility of the interface, which can be Internal or External;*
- *card: refers to the cardinality of the interface, which is Singleton or Collection;*
- *contig: characterizes the interface contingency, which may be Optional or Mandatory;*
- *sign: returns the interface signature.*

Definition 4. *A component binding b is defined with:*

$$b = \prec itfc - clt, itfc - srv \succ$$

where:

- *itfc-clt: refers to the invoking interface, and can be Client or Sink;*
- *itfc-srv: refers to the service interface, and may be Server or Source.*

2) Reconfiguration:

Definition 5. *Let be a configuration Cg_1 of a system S. We define a reconfiguration R of S, being in the configuration Cg_1, as an ordered set of primitive operations applied on Cg_1 :*

$$R = op_1, op_2, ..., op_n, n \geq 1$$

where $op_i, i = 1..n$, is one of the following reconfiguration operations:

1) *Delete a component*
2) *Add a component*
3) *Replace a component*
4) *Delete a binding*
5) *Add a binding*

The resulted configuration after application of R is denoted Cg_2.

We denote this by:

$$Cg_1 \overset{op}{\rightarrow} Cg_2$$

3) Predefined functions: To be able to specify constraints required for performing properly a reconfiguration, we need a set of predefined functions. For this objective, we propose the following functions:

1) CFather(cp) : returns the parent of the component cp;
2) CInterfaces(cp) : returns the interfaces list of the component cp;
3) CType(cp) : returns the type of the component cp;
4) IComponent(i) : returns the owner of the interface i;
5) IType(i) : returns the type of the interface i.

B. Constraints

To ensure the correction of a reconfiguration R applied on a system S, we define two sets of constraints:

- Constraints on primitive reconfiguration operations : Should be checked after each primitive operation.
- Global constraints : should be checked after the whole reconfiguration.

In the following, we specify these two sets of constraints.

1) Constraints on primitive reconfiguration operations: Let op be a primitive reconfiguration operation, applied on a configuration Cg_1 of a system S, resulting in the configuration Cg_2, where :

- $Cg_1 = \prec C_1, I_1, B_1 \succ$
- $Cg_2 = \prec C_2, I_2, B_2 \succ$

We denote this by:

$$Cg_1 \overset{op}{\rightarrow} Cg_2$$

In the following, we consider :

- A component : $cp = \prec name, granul, statut \succ$
- A binding : $b = \prec iclt, isrv \succ$

Primitive reconfiguration operations, applied on components cp and cp', are denoted as follows:

1) Delete a component cp :

$$del\_comp(cp)$$

2) Add a component cp :

$$add\_comp(cp)$$

3) Replace a component cp by anpther cp':

$$Repl\_comp(cp, cp')$$

4) Delete a binding b :

$$del\_bdg(b)$$

5) Add a binding b :

$$add\_bdg(b)$$

Table I gives the required constraints to be satisfied after each reconfiguration operation.

 ISBN: 978-1-61208-304-9

TABLE I: CONSTRAINTS ON PRIMITIVE RECONFIGURATION OPERATIONS

Operation	*Constraints*
del_comp(cp)	1) $C_2 = C_1 - cp$ 2) $I_2 = I_1 - CInterfaces(cp)$ 3) $\forall$ i $\in CInterfaces(cp), i \notin I_2$
add_comp(cp)	1) $C_2 = C_1 \cup cp,\ cp \notin C_1$ 2) $I_2 = I_1 \cup CInterfaces(cp)$ 3) $\forall i \in I_1, \exists j \in I_2\ tq:\ i.itfc = j.itfc$ 4) $\forall i \in CInterfaces(cp), i \notin I_1$
Repl_comp(cp,cp')	• $CType(cp)$ *is a subtype of* $CType(cp')$
del_bdg(b)	1) $B_2 = B_1 - b$ 2) $b \in B_1$
add_bdg(b)	1) $B_2 = B_1 \cup b$ 2) $b \notin B_1$ 3) $\exists\ b.iclt \in I_1 \wedge b.isrv \in I_1$ 4) $b.iclt.card$=$SINGLETON \Rightarrow$ $\forall$ b'$<$ $iclt', isrv'$ $>\in B_1, iclt' \neq iclt$ 5) $b.isrv.card$=$SINGLETON \Rightarrow$ $\forall$ b'$<$ $iclt', isrv'$ $>\in B_1, isrv' \neq isrv$

2) Global constraints: Let R be a reconfiguration that will be applied to a configuration Cg_1 of a system S, giving as a result a configuration Cg_2:

$$Cg_1 \overset{R}{\to} Cg_2$$

$$with:\ R = op_1, op_2, ..., op_n, n \geq 1$$

We specify the following constraints, which must be satisfied by Cg_2 :

1) $\forall$ b $\in B_2$, b.iclt.role = Client / Sink $\wedge$ b.isrv.role = Server / Source
2) $\forall$ b $\in B_2$, (b.iclt.contig = Mandatory) $\Rightarrow$ (b.isrv.contig = Mandatory)
3) $\forall$ b, b' $\in B_2$, b.iclt $\neq$ b'.iclt
4) $\forall$ b $\in$ B_2, (CFather(IComponent(b.iclt)) = CFather(IComponent(b.isrv)))$\vee$ (b.iclt.visib = Internal $\wedge$ IComponent(b.iclt) = CFather(IComponent(b.isrv))) $\vee$ (b.isrv.visib = Internal $\wedge$ CFather(IComponent(b.iclt)) = IComponent(b.isrv))
5) $\forall$ i $\in I_2$ (i.role = Client $\wedge$ i.contig = Mandatory $\Rightarrow$ $\exists$! b $\in B_2$ tq: b.iclt = i)
6) $\forall$ b $\in B_2$, IType(b.isrv) $\subseteq$ IType(b.iclt)

C. Consistency of a configuration

Theorem 1. *A reconfiguration R, applied to a configuration Cg_1 of a system S, is* valid *if the resulting configuration Cg_2 satisfies all constraints defined on primitive reconfiguration operations and global constraints.*

Theorem 2. *A configuration Cg_i of a system S is* consistent *after a reconfiguration R if R is* valid*.*

V. ILLUSTRATION

To illustrate our approach, we use a navigator application similar to Mozilla already used in [20]. In such applications, components are usually equipped with an install manifest in XML format, allowing, among other things, to deliver the information needed to manage the version compatibility between components.

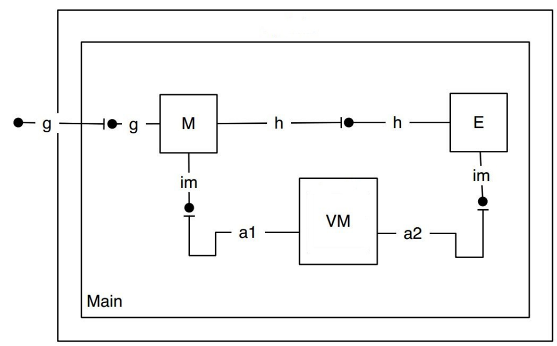

Fig. 1: Initial configuration

So, the architecture of the application consists of a composite component $MAIN$ composed of three primitive components (Figure 1):

1) M, the main application (e.g., Firefox);
2) E, an already installed plugin;
3) VM, a version manager component.

Each of the components M and E have an interface h with a signature H, being respectively a client and server interface. They also each have a server interface im of signature $InstallMf$. M has an additional server interface g of signature G, being the main interface exported to the global external interface of the application.

The Main composite exports business methods from M and supplies update, a control method implementing the upgrade operation. This method looks for a component with same id as E, having a more recent version and being compatible with M. In case of success, it replaces E with the new component.

Based on our formalization, we specify the initial configuration of Figure 1 as follows:

$$Cg_1 = \prec C_1, I_1, B_1 \succ$$

where:

- $C_1 = (Main, M, VM, E)$
- $I_1 = (Main.g, M.g, M.im, VM.a1, VM.a2, E.im, E.h)$
- $B_1 = (b1, b2, b3, b4)$

where:

- M = (M, Primitive, Started)
- VM = (VM, Primitive, Started)
- E = (E, Primitive, Started)
- Main.g = (Main.g, Server, External, Singleton, Optional, G)
- M.g = (M.g, Server, External, Singleton, Optional, G)
- M.h = (M.h, Client, External, Singleton, Optional, H)
- M.im = (M.im, Server, External, Singleton, Mandatory, InstallMF)
- VM.a1 = (VM.a1, Client, External, Singleton, optional, InstallMF)
- VM.a2 = (VM.a2, Client, External, Singleton, Optional, InstallMF)
- E.h = (E.h, Server, External, Singleton, Optional, H)
- E.im = (E.im, External, Singleton, Mandatory, InstallMF)
- b1 = (Main.g, M.g)
- b2 = (VM.a1, M.im)
- b3 = (VM.a2, E.im)
- b4 = (M.h, E.h)

When applying on this configuration a reconfiguration R, which removes the plugin E, we model this by the following reconfiguration R :

$$R = op_1, op_2, op_3$$

where:

- $op_1 : Del\_comp(E)$,
- $op_2 : Del\_bdg(b3)$,
- $op_3 : Del\_bdg(b4)$.

This resulted configuration is valid because it provides a new consistent configuration (given in Figure 2), which is defined as follows:

$$Cg_2 = \prec C_2, I_2, B_2 \succ$$

where:

- C_2 = (Main, M, VM)
- I_2 = (Main.g, M.g, M.im, M.h, VM.a_1, VM.a_2)
- B_2 = (b1)

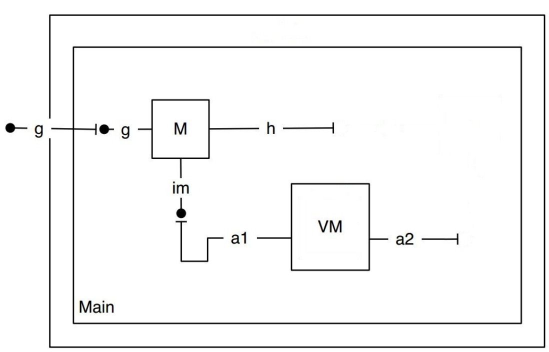

Fig. 2: Resulting configuration after reconfiguration

where:

- M = (M, Primitive, Started)
- VM = (VM, Primitif, Started)
- Main.g = (Main.g, Server, External, Singleton, Optional, G)
- M.g = (M.g, Server, External, Singleton, Optional, G)
- M.h = (M.h, Client, External, Singleton, Optional, H)
- M.im = (M.im, Server, External, Singleton, Mandatory, InstallMF)
- VM.a1 = (VM.a1, Client, External, Singleton, optional, InstallMF)
- VM.a2 = (VM.a2, Client, External, Singleton, Optional, InstallMF)
- b1 = (Main.g, M.g)
- b2 = (VM.a1, M.im)

By checking all defined constraints, we can say that R is valid. So, the new configuration $Cg2$ is consistent starting from the fact that $Cg1$ is consistent.

VI. CONCLUSION

In this paper, we presented a new formalism for checking consistency of dynamic reconfigurations of general component-based systems. For this purpose, we introduced formal concepts for modelling a component-based configuration and reconfiguration operations. We also defined required constraints that must be satisfied by the new configuration resulting after applying reconfiguration, to ensure consistency of the system.

Our approach can be instanciated to any existing component model, allowing thus genericity of the formalism.

Work is in progress to achieve automation of the proposed approach, by providing a toolbox based on the FOCALIZE programming environment [21]. This latter is based on a functional programming language with object-oriented features and allows to write formal specifications and proofs of designed programs. Proofs are build using the automated theorem prover Zenon [22] and Coq proof-assistant [23]. Future work also include modeling CBS before and after reconfiguration to allow quantitative analysis of CBS.

REFERENCES

[1] C. Szyperski, *Component software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 2002, vol. 2nd Edition.
[2] Sun Microsystems, "EJB 3.0 specification," http://www.oracle.com/technetwork/java/docs-135218.html, jul 2007.
[3] Object Management Group, "Corba component model (ccm) (CORBA) - specification, version 3.1, part 3: CORBA components," http://www.omg.org/spec/CORBA/3.3/Interoperability/PDF (November 2008), 2008.
[4] E. Bruneton, "Fractal tutorial," http://fractal.objectweb.org/tutorials/fractal/index.html (September 12 2003), September 2003.
[5] M. Leger, "Reliability of dynamic reconfigurations in component architectures," Ph.D. dissertation, Superior National School of Mines of Paris, 19 mai 2009.
[6] R. M. Vincenzo Grassi and A. Sabetta, "From design to analysis models: a kernel language for performance and reliability analysis of component-based systems," vol. 80, no. 11, 2005, pp. 25–36.
[7] O. M. Group, "(uml) unified modeling language (3rd release)," November 2004.
[8] Object Management Group, "UML unified modeling language (3rd release)."

[9] D. Freeman, *Markov chains*, Springer-Verlag, Ed. Springer-Verlag, 1983.
[10] L. Kleinrock, *Queueing systems. Volume I : Theory*, Wiley-Interscience, Ed. New-York: Wiley-Interscience, 1975.
[11] V. Grassi, R. Mirandola, and A. Sabetta, “A model-driven approach to performability analysis of dynamically reconfigurable component-based systems,” pp. 103–114, 2007.
[12] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe, “An automated formal approach to managing dynamic reconfiguration,” in *Automated Software Engineering, 2006. ASE ’06. 21st IEEE/ACM International Conference on*, september 2006, pp. 37–46.
[13] J. Hillman and I. Warren, “An open framework for dynamic reconfiguration,” in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 594–603.
[14] D. Jackson, “Alloy : a lightweight object modelling notation,” vol. 11, no. 2, Novembre 2002, pp. 256 – 290.
[15] L. T. David P.-C., “An aspect-oriented approach for developing self-adaptive fractal components,” 2006, pp. 82–97.
[16] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani, “The fractal component model and its support in java,” vol. 36, no. n. 11-12, 2006, pp. 1257–1284.
[17] M. Simonot and M. Aponte, “Formal modeling of control with fractal,” CEDRIC laboratory, CNAM-Paris, France, Tech. Rep. CEDRIC-08-1590, 2008, http://cedric.cnam.fr/index.php/publis/article/view?id=1590.
[18] V. Benayoun, “Fractal components with dynamic reconfiguration : formalization with focal,” 2008, http ://reve.futurs.inria.fr/.
[19] C. Szyperski, “Component technology - what, where, and how?” in *Proc. 25th Int. Conf. on Software Engineering*. IEEE, May 3–10 2003, pp. 684–693.
[20] M. Simonot and V. Aponte, “A declarative formal approach to dynamic reconfiguration,” pp. 1–10, 2009.
[21] INRIA and LIP6, “The focalize essential,” 2005, http://focalize.inria.fr/.
[22] D. D. R. Bonichon and D. Doligez, “Zenon : An extensible automated theorem prover producing checkable proofs,” vol. 4790, 2007, pp. 151–165.
[23] Y. Bertot and P. Casteran, *Interactive Theorem Proving and Program Development Coq Art: The Calculus of Inductive Constructions*. Addison-Wesley, 2004.

 ISBN: 978-1-61208-304-9