Reliability of dynamic reconfigurations in component-based software systems

Marc Léger[1], Thierry Coupaye[1], and Thomas Ledoux[2]

[1] France Telecom R&D
28, chemin du Vieux Chêne
F-38243 Meylan
{marc.leger, thierry.coupaye}@orange-ftgroup.com
[2] OBASCO Group, EMN / INRIA, LINA
Ecole des Mines de Nantes
4, rue Alfred Kastler
F-44307 Nantes Cedex 3
thomas.ledoux@emn.fr

Abstract. This article is an analysis based on our experience with the Fractal component model of the need of reliability for dynamic reconfigurations in component based systems. We make a proposal to ensure this reliability, which can applied to concurrent reconfigurations. We started from the definition of ACID properties in the context of component models and we propose to use integrity constraints to define system consistency and transactions for guaranteeing the respect of these constraints at runtime. To deal with concurrency, we have to detect potential conflicts when composing reconfiguration operations.

1 Introduction

Dynamic reconfigurations in component-based software applications [MK96] are central to promissing approaches like autonomic computing [KC03]. There are many motivations to introduce modifications in a system at runtime: correction of security flaws or functional bugs, improvement of systems (e.g., performance optimizations), or adaptions to execution context changes.

Thanks to properties of component models like loosely coupling, reconfigurations can rely on component-based architectures [OMT98]. However, runtime modifications can let the system in an inconsistent state. From a structural point of view, the architecture of the system once reconfigured can be not in conformity with the component model or eventually system specific constraints (e.g. architectural invariants) anymore. From a functional point of view, a reconfiguration must not perturb the execution of the system (i.e., functional and non functional aspects need to be synchronized). Furthermore, in case of concurrent reconfigurations, reconfiguration must be synchronized between themselves.

In this paper, we focus to the reliability of runtime adaptations and we chose to base our work on the Fractal component model [BCL+04] because of its support of dynamic and opened reconfigurations. In our approach, we

tried to define each of the ACID properties [TGGL82] in the specific context of component-based systems an show how it can solve this reliability problem during adaptations. These properties are unifying concepts of transactions for distributed computation used essentially for supporting concurrency and recovery. We specify the consistency property by using integrity constraints about system structure and state. An example of a structural constraint at the level of component model is cycle-free component structure. Moreover we must avoid wrong execution flow of reconfiguration operations according to their semantics to ensure the isolation property.

This paper is organized as follows. Section 2 is an overview of dynamic reconfigurations in component models, with a focus on Fractal, and it shows what problems it raises regarding reliability. Then section 3 describes how transactions combined with integrity constraints can be a solution to these problems. Finally section 4 presents some related works before concluding in section 5.

2 The need of reliability for dynamic reconfigurations in component-based systems

2.1 Dynamic reconfigurations in component models

Dynamic reconfigurations allow modifications of a part of a system during its execution without stopping it entirely to keep the system partly available. Actually, maximization of the availability time is essential for some systems like entreprise application servers. Dynamic reconfigurations can involve every manageable element defined in the component model and reified at runtime, they can be:

- structural (e.g., addition or removal of elements like components, interfaces etc. and interconnection modifications with bind unbind operations),
- behavioral (e.g., lifecycle modification used to synchronize component activity with the rest of the system),
- linked to component deployment (e.g., component instantiation, destruction, migration),
- linked to component state (e.g., change of component attribute values),

Fractal [BCL$^+$04] is a recursive component model with sharing and reflexive control. It is based on classic concepts of component (as a runtime entity), interface (an interaction point between components expressing provided and required services) and binding (a communication channel between component interfaces). A component consists of a membrane which can show and control a causaly connected representation of its encapsulated content. An Architecture Description Language (Fractal ADL [Fra]) is used to specify component configurations and there is notably a Java implementation of the model, Julia. Several controllers are defined to control bindings, the hierarchical structure, component lifecycle, attributes and names, but other controllers can be user-defined.

Operations in controllers constitute primitive reconfiguration operations and do either introspection or intercession (modifications) in the system. To compose

operations, we consider sequences or parallel executions of intercession operations with conditions expressed by means of introspection operations in component configurations. An example of composite reconfiguration is component hotswap, a mechanism used to update a system where an old version of a component is replaced by a new one. In Fractal, this reconfiguration is composed of a sequence of several primitive reconfiguration operations, it implies to stop the component, unbind all its interfaces, remove it, add the new instantiated component, bind its interfaces and start it (a state transfert operation is used in case of stateful component).

2.2 The reliability problem with dynamic reconfiguring applications

A first problem when modifying a system at runtime is the synchronization between reconfigurations and the functionnal execution of the system. Actually, the part of the system which is modified could be unavailable for functional execution during the reconfiguration time. To take the hotswap example with a stateful component, calls on the old component must be blocked until a a "quiescent state" [KM90] is reached, then the state must be transfered, finally previous calls are forwarded towards the new component.

A second problem at the model level is about consistency violation by reconfigurations. First of all, we must make clear what exactly consistency is for component-based systems. Component models and application models should define what this consistent system is, especially in term of structure. For instance, we may want to add a structural constraint about the number of subcomponents of a composite component. In Fractal, the specification of the component model is not always sufficient and we want to express some integrity constraints on systems. So we must ensure the conformity of the system to the model and constraints after reconfigurations.

The third and last problem we identified is linked to the composition of reconfiguration operations. A prerequisite is the separation of concerns between the functional part and the control part of systems. Then separation between introspection operations and intercession operations must be explicit. Once these operations have been identified, the semantics of reconfiguration operations implies there can be some conflicts between them in case of compostion and for synchronization between several reconfigurations (e.g., in Fractal it is mandatory to unbind all component interfaces before removing the component from its super-component).

3 A transactional approach to ensure reliable reconfigurations

3.1 ACID properties in the context of dynamic reconfigurations

We think that well-defined transactions associated with structural and behavioral constraints verification is a means to guarantee the reliability of reconfigurations in component models, i.e. to solve problems we identified in the section

2.2. As any reconfiguration operation could lead the system to an inconsistent state, each reconfiguration must always be included in a transaction. In this context, we define the meaning of ACID properties as follows:

- **Atomicity**: either all happen or none happen, that is to say either the system is reconfigured or it is not. A reconfiguration transaction can be a single primitive reconfiguration operation or a more complex operation composed of several operations. Each reconfiguration operation must specify its reversible operation. Thus if a reconfiguration transaction goes badly and is rollbacked, it is possible to come back in a previous stable state by undoing operations. Transactions demarcation is either programmed in the language or automatic (a reconfiguration script corresponds to a transaction).
- **Consistency**: a transaction must be a correct transformation of the system state. So the reconfigured application must be conform to the component model and application specific constraints. That is to say consistency is given by integrity constraints essentially architectural invariants. A reconfiguration transaction can be commited only if the resulting system respects the constraints. Other faults like software and hardware failures (network and machines) are the responsibility of the commit protocol (e.g., 2 phase commit protocol).
- **Isolation**: several reconfiguration transactions are independant and any schedule of reconfiguration operations must be equivalent to their serialization. The scheduling must respect the operation semantics and conflicts. This property relies on the knowledge of the semantics of reconfiguration operations.
- **Durability**: once a reconfiguration completes with success (commit), the new state is persistent. For every transaction, operation are logged in a journal so that reconfigurations can be redone in case of failure. The application state (architecture and component state) is periodically checkpointed basically with ADL dumps and component state is saved in databases. So any component can be recovered in its last stable state resulting from the last successful reconfiguration. However, the only functional state we capture is the state which is well identified in the component model and is saved only at commit time of reconfigurations because we don't want to impose transactions at the functional level.

Only the first problem presented in 2.2 is not completely adressed by our approach because we do not fully modelise the functional execution flow of systems, we relies on the implementation of the component lifecycle operations with interceptors on component interfaces to realize the synchronization. A solution to the synchronization problem is to apply the hotswap protocol proposed in [KM90]. The guarantee we can bring is that the order of operations in the protocol is respected. Among the ACID properties we will especially focus in the following sections on two properties: consistency and isolation.

3.2 Integrity constraints to ensure system consistency

In our proposal, system consistency relies on integrity constraints and we want to express these constraints both at the application and at the model level. An integrity constraints is essentially a predicate which concerns the validity of an assembly of architectural elements but it can also concern component state. Examples of such constraints at the component model level are hierarchical integrity (bindings between components must respect the component hierarchy) or cycle-free structure (a component cannot contain itself to avoid infinite recursion). On the other hand, application specific constraints are used to specify invariants on a given system either on component types or directly on component instances designed by their names. Invariants can concern for example cardinality of sub-components in a super-component, two component interfaces which can never be unbound etc.

In an open world where reconfigurations are not anticipated at compile time, some component models like Fractal are relying on reflexive architectures to dynamically reconfigure systems by means of a runtime mapping between the system which is really executed and its model. So integrity constraints verified on the model will be also valid in the system. We represent the Fractal component model as a typed graph and then each fractal-based application is also a graph which is an instance of this typed graph. The instance graph is a more formal representation of the system provided at runtime by the reflexivity of the component model and is used to navigate in runtime applications. The vertexes are elements from the component model: components, functional interfaces, controllers, attributes and operations. The edges represent relations between the elements: composition links, binding links etc. Then the instance graph must always be well-typed regarding to the typed graph (i.e., conform to the component model) and the instance graph must respect integrity constraints. Therefore contraints at the model level can be specified on the typed graph and others on the instance graph. As the model is extensible and new user-defined controllers can be added, graphs should be also easily extensible in terms of elements and relations.

To express integrity constraints, we propose to use a DSL based on an extension of the query language in Fractal configurations FPath [DL06] to transform it into a real constraint language "à la OCL" [OCL05]. An advantage of the FPath language is that it can navigate both in the ADL and in the runtime system and it is already based on a graph representation of the system during execution. The constraint language must just have introspection capacity without side effects on the system. We want to express invariants, preconditions and postconditions in this language and we want notably to have quantifiers, collection operations and filters. The following basic example is a structural invariant constraint at the application level expressed in FPath (with its XPath 1.0 syntax like) where the component designed by the variable c can never be shared (it can only have one parent at the same time):

```
size(c/parent::*)=1
```

Constraints must be checked both at compile time on the component static configuration and at runtime. We consider checking constraints as far as possible

before applying the reconfiguration on the system, eventually by code analysis of a dedicated reconfiguration language like FScript [DL06]. Constraints can also be checked either directly during the execution of the reconfiguration of the real system or by simulation on a local copy of the representation of the system (i.e., the instance graph) so as to limit the effect on the system in case of constraint violation.

3.3 Isolation of reconfigurations to support concurrency

We take the hypothesis that not only application components are distributed but also administrators. Furthermore, reconfiguration initiators are either humans (interactive reconfigurations) or the system itself (the system is able to auto-reconfigure). Concurrency in reconfigurations comes from the fact that one administrator can explicitely want to execute some operations in parallel, or several administrators can reconfigure the same system at the same time. The reconfiguration scheduler can also detect when it can launch parallel reconfiguration tasks to optimize the reconfiguration process.

As seen in section 2.1, reconfiguration operations are composable but all compositions are not valid. In Julia, operation semantics is hidden in controller implementations and so we want to make it explicit and we want eventually to be able to change it and to specify new primitive operations. So we need to express operation semantics in terms of preconditions and postconditions with our constraint language presented in section 3.2. We distinguish two types of conflicts between operations: parallel conflicts and execution dependencies. For two given reconfigurations *R1* and *R2* executed on the same system, a parallel conflict occurs if *R1* and *R2* modify the same manageable elements in the system model (e.g. bind and unbind operations). An execution dependency occurs if *R1* either need *R2* to be executed first (e.g. stop before unbind)or if *R1* cannot be executed after *R2*. That is to say *R2* postconditions cover or not *R1* preconditions.

```
// Example of a precondition for removing a component
operation: void removeSubComponent(Component sub);
preconditions :
// all interfaces of the sub-component are unbound (. is the current node)
not(exists(sub/interface::*[not(bound(.))]));
```

For concurrency management, we propose a pessimistic approach with locking. Our locking algorithm is based on operation semantics to avoid inconsistent operation compositions. We see two different possibilities for the locking algorithm. The first one is to lock directly reconfiguration operations. That is to say, either conflicts between operations are automatically calculated thanks to its preconditions and postconditions or it must define the operations with which it is in conflict. The second one is to use a modified DAG locking algorithm on our instance graph defined in 3.2. Then the lock granularity is defined by the

manageable elements in the graph representation and for example a lock acquisition on a component also locks all its interfaces and every operations in each interfaces.

Another approach to locking is to constrain the execution order of reconfiguration operations. We propose to use a simple language inspired of behavior protocols in [PV02] to describe the desired execution order of reconfiguration operations, what we call behavioral reconfiguration constraints. The protocol compliance is checked at runtime by intercepting reconfiguration calls.

4 Related work

Many works on ADLs follow a static approach to check consistency of component-based architectures by compilation but only a few are interested in dynamic analysis of this consistency. We will focus here on other reflective component models which allow non anticipated (also called ad-hoc) reconfigurations.

FORMAware [MBC04] is relatively close to our work. This framework to program component-based application gives the possibility to constrain reconfigurations with architectural style rules. A transaction service manages the reconfiguration by stacking operations. The main difference with our proposal is our integrity constraints are more flexible than styles and they can be applied to every element of our component model. Moreover we define more formally reconfiguration operations to identify conflicts between them, our locking algorithm is then more precise than a simple lock on components and we consider introspection operations as reconfiguration operations.

Plastik [BJC05] is the integration of the OpenCOM component model and the ACME/Armani ADL. As in our solution, architectural invariants can be checked on ADL configuration or at runtime and constraints are expressed at the style level and at the instance level. However, reconfiguration cannot be generic composite reconfigurations with model elements in parameters and the execution, the operation semantics is not explicit and not extensible and the order of reconfiguration operation cannot be constrained as we can do with reconfiguration protocols.

5 Conclusion

Dynamic reconfiguration in component-based systems raises reliability problems, especially in open systems in which they are not anticipated. In this article, we identified the three following global problems based on our experience with the Fractal component model: synchronization between reconfiguration and the functional execution of systems, consistency regarding component and application models, and synchronization between reconfiguration operations. We focused more on the two last problems: the first one concerns conformity at runtime of systems with constraints and models, the second one deals with the validity of composition of reconfiguration operations.

We propose to use integrity constraints to define consistency for dynamic reconfigurations and to include these reconfigurations in transactions. We build a graph representation of our application at runtime thanks to the reflexivity of the Fractal component model and use a constraint language on this graph. Moreover we want to detect execution conflicts between reconfiguration operation in order to be able to compose them with reliability with eventually the specification of reconfiguration protocols. We are currently implementing this proposal in Julia, a Java implementation of the Fractal model.

References

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *CBSE*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[BJC05] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In Ronald Morrison and Flávio Oquendo, editors, *EWSA*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[DL06] Pierre-Charles David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceedings of the 5th Fractal Workshop at ECOOP 2006*, Nantes, France, July 2006.

[Fra] Fractal ADL. *http://fractal.objectweb.org/fractaladl*.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KM90] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[MBC04] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. Supporting adaptable distributed systems with formaware. In *ICDCSW '04: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*, pages 320–325, Washington, DC, USA, 2004. IEEE Computer Society.

[MK96] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM Press.

[OCL05] OCL 2.0 Specification. *http://www.omg.org/docs/ptc/05-06-06.pdf*, 2005.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[PV02] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.

[TGGL82] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*, 7(3):323–342, 1982.