Component Deployment Evolution Driven by Architecture Patterns and Resource Requirements

Didier Hoareau, Chouki Tibermacine

Component Deployment Evolution Driven by Architecture Patterns and Resource Requirements

Didier Hoareau and Chouki Tibermacine

VALORIA Lab., University of South Brittany, France
{Didier.Hoareau,Chouki.Tibermacine}@univ-ubs.fr

Abstract. Software architectures are often designed with respect to some architecture patterns, like the pipeline and peer-to-peer. These patterns are the guarantee of some quality attributes, like maintainability or performance. These patterns should be dynamically enforced in the running system to benefit from their associated quality characteristics at runtime. In dynamic hosting platforms where machines can enter the network, offering new resources, or fail, making the components they host unavailable, these patterns can be affected. In addition, in this kind of infrastructures, some resource requirements can also be altered. In this paper we present an approach which aims at dynamically assist deployment process with information about architectural patterns and resource constraints. This ensures that, faced with disconnections or machine failures, the runtime system complies permanently with the original architectural pattern and the initial resource requirements.

1 Introduction

When we design software architectures, we often make use of architecture patterns, like for example the pipe and filter, the client and server, peer-to-peer pattern, etc. These vocabularies of recurrent solutions to recurrent problems[1] are the guarantee of some quality attributes in the designed system. These quality attributes include maintainability, portability, reliability and performance. Starting from these high-level design documents (architecture descriptions), we can produce low-level implementation entities that will be deployed.

One of the characteristics of emerging distributed platforms is their dynamism. Indeed, such dynamic platforms are not only composed of powerful and fixed workstations but also of mobile and resource-constrained devices (laptops, PDAs, smart-phones, sensors, etc.). Due to the mobility and the volatility of the hosts, connectivity cannot be ensured between all hosts, e.g. a PDA with a wireless connection may become unaccessible because of its range limit. As a consequence, in a dynamic network, partitions may occur, resulting in the fragmentation of the network into islands. Machines within the same island can communicate whereas, no communication is possible between two machines that

[1] With analogy to design patterns but at a more coarse-grained level of abstraction.

are in two different islands. Moreover, as some devices are characterized by their mobility, the topology of islands may evolve.

Dynamism in the kind of networks we target is not only due to the nature of the devices but also to their heterogeneity making difficult to base a deployment on resource's availability. When deploying component-based software in dynamic distributed infrastructures it is required that the deployed system complies permanently with its corresponding architecture pattern(s). By taking advantages of changes in the environment (e.g. availability of a required resource), the initial deployment can evolve but any reconfiguration must respect architectural choices. This makes the running system benefit from the targeted quality attributes, and more particularly those which are dynamically observed, like performance or reliability.

In this paper, we present an approach to drive component deployment and component deployment evolution in this kind of dynamic networks, based on information about architecture patterns and resource requirements. This approach uses two kinds of constraints: the first one represents patterns and resource requirements formalisation; the second one corresponds to the result of transforming the former constraints into run-time ones. These run-time constraints are checked dynamically and are used to drive component deployment and component deployment evolution.

In the next section we present how we can formalize architecture patterns and resource requirements using a constraint language, and we illustrate this formalization by a short example of a client/server pattern. We present in section 3, the deployment process and the resolution mechanisms of these constrained component-based software in dynamic infrastructures. Before concluding, we present some related work in section 4.

2 Formalization of Architectural Decisions with ACL

In order to make explicit architectural decisions, we proposed ACL, an Architecture Constraint Language [11]. Architectural decisions are thus formalised as architecture predicates which have as a context an architectural element (component, connector, etc.) that belongs to an architecture metamodel. ACL is a language with two levels of expression. The first level encapsulates concepts used for basic predicate-level expression, like quantifiers, collection operations, etc. It is represented by a slightly modified version of UML's OCL [9], called CCL (Core Constraint Language). The second level embeds architectural abstractions that can be constrained by the first level. It is represented by a set of MOF architecture metamodels. Architectural constraints are first-order predicates that navigate in a given metamodel and which have as a scope a specific element in the architecture description. Each couple composed of CCL and a given metamodel is called an ACL profile. We defined many profiles, like the ACL profile for xAcme (which is an XML extension of Acme ADL [3].), for UML 2, for OMG's CORBA Components [8] (CCM) or the profile for ObjectWeb's Fractal [1].

2.1 Architectecture pattern description

Suppose that we have developed, at architecture design-time, a component software that represents a company printing system. We would like to automate the installation and the reconfiguration of this system to all company employees. This printing system is organised according to the client/server pattern. The printing service is based on a ServerPrinter which receives print jobs from ClientPrinter. The client/server pattern is characterized by the following constraints: i) there is no direct communication between ClientPrinters, ii) each ServerPrinter can accept jobs from at most 10 different clients, and iii) a ClientPrinter can use at most two ServerPrinters. These first two constraints can be described using ACL profile for Fractal as following:

```
context ClientServer:CompositeComponent inv:
ClientServer.binding->forAll(b|b.client.component.kind
<> b.server.component.kind)
and
ClientServer.subComponents->select(c|c.kind='Server')
.interface->oclAsType(Server).binding->size() <= 10
```

These constraints navigate in the MOF metamodel of Fractal ADL which is presented in Figure 1. This metamodel abstracts components, which can be composite or primitive. Composite (or hierarchical) components are entities which have an explicit description of their internal parts. Primitive (or atomic) components are directly implemented by an object class. Components express their functionalities and requirements through respectively, server and client interfaces. In addition, controller interfaces embed non-functional specifications, such as predefined operations which manage the lifecycle or the contents of a given component. A composite component specifies also a set of bindings which are simple method invocation connectors. These bindings are attachments between client and server interfaces. Bindings can represent either hierarchical or assembly connectors (with analogy to UML's delegation and assembly connectors). Hierarchical connectors bind interfaces of composite components to interfaces of their sub-components. Assembly connectors bind interfaces of components of the same level of hierarchy.

2.2 Resource and location requirements description

In addition to these architecture design constraints, the deployment of each component is governed by some resource and location requirements. Indeed, at design time, we are unlikely to know the machines that are involved in the deployment and thus where to deploy each component. However, one can define for each component its requirements in term of resources, that is, the characteristics of the machines that will host the component. For example, a ServerPrinter must be hosted by a machine that has at least 512 MB of free memory, a CPU scale greater than 1 GHz (1000 MHz) and that is connected to a printer.

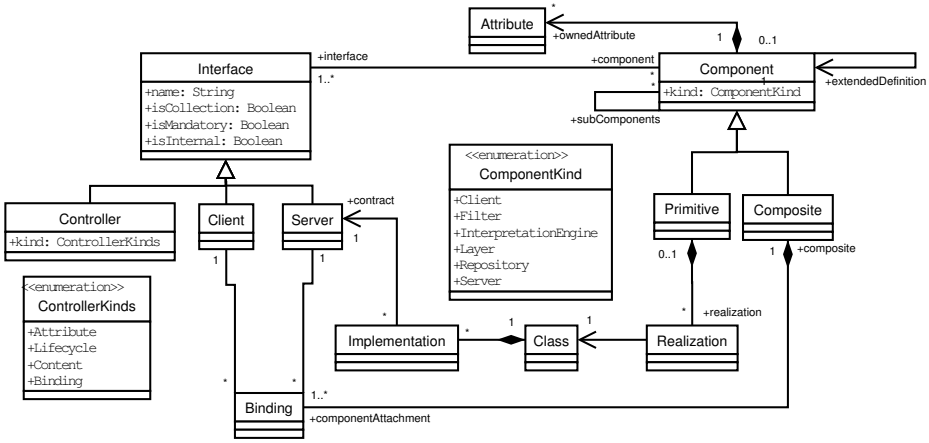

Fig. 1. A MOF metamodel of Fractal ADL

Resource constraints can be defined using an ACL profile (i.e. a CCL and a metamodel), called R-ACL (Resources-ACL). R-ACL integrates in its metamodels concepts related to system resources and their properties[2]. Resource constraints introduced above and related to ServerPrinters components are described in R-ACL as following:

```
context ServerPrinter:Component inv:
ServerPrinter.resource->oclAsType(Memory).free >= 512
and
ServerPrinter.resource->oclAsType(CPU).processors
->select(cpu:CPU_Model|cpu.speed > 1000)->size() >= 1
and
ServerPrinter.resource->oclAsType(Devices)
->select(printer:Printer)->size() >= 1
```

As discussed above, these constraints navigate in the resources metamodel, but have as a scope a specific architectural element (ServerPrinter component).

Besides resource constraints, it is sometimes required to control the placement of the components, especially when several machine can host the same component. For example in the Client/Server system we designed, we would require that for reliability reasons (redundancy at the server side), all ServerPrinters have to be located on distinct hosts. The following listing illustrates this constraint expressed in R-ACL.

```
context ClientServer:CompositeComponent inv:
ClientServer.subComponent->select(c1,c2:Component|c1.kind='Server'
and c2.kind='Server' and c1.location.id <> c2.location.id)
```

[2] The resources metamodel is not presented in this paper due to space limitations.

3 Constrained Components' Deployment in Dynamic Infrastructures

When the choice of the placement of every component has to be made, the initial configuration of the target platform may not fulfil all resources' requirements of the application and some needed machines may not be connected. We are thus interested in a deployment that allows the instantiation of the components as soon as resources become available or new machines become connected. We qualify this deployment as *propagative*. We propose a general framework to guarantee the designed architecture and its instances for each deployment evolution.

We present first the requirements of a deployment driven by pattern and resource specifications. Then, we detail the deployment process and the resolution of constraints in dynamic environments.

3.1 From architectural constraints to runtime constraints

At design time, we are unlikely to know what are the machines that are involved in the deployment and thus what are their characteristics. Hence, a valid configuration of the client/server pattern presented in section 2, can only be computed at runtime. A valid configuration is a set of component instances, interconnected and for which, a target host has been chosen. Every architectural constraint (e.g. on bindings or number of instances) has to be verified and the selected hosts must not contradict the resource and location constraints.

Our approach consists in manipulating all the architectural and resource constraints at runtime in order to reflect the state of the deployed system with respect to these constraints. The reified constraints are generated automatically from the R-ACL constraints and correspond to a constraint satisfaction problem (CSP). In a CSP, one only states the properties of the solution to be found by defining variables with finite domains and a set of constraints restricting the values that the variables can simultaneously take. The use of solvers such as Cream[3] can then be used to find one or several solutions. The CSP that corresponds to our patterns consists of the following constraints:

C1 the number of instances allowed for each component
C2 the resource constraints (e.g. $Mem.free \geq 512$)
C3 the location constraints (e.g. $x \neq y$)
C4 a binding constraint between every component that can be bound
C5 the number of outgoing bindings allowed on a client interface
C6 the number of incoming bindings allowed on a server interface

Each C_i corresponds to a set of constraints. As we will detail below, these sets are sufficient to generate a valid configuration regarding to an architectural pattern. The deployment process that is presented in the next section relies on these constraints in order to build a mapping between the component instances and the hosts of the target platform.

[3] `http://kurt.scitec.kobe-u.ac.jp/~shuji/cream/`

3.2 Deployment process

When dealing with dynamic networks where partitions may occur and hosts availability has to be faced with, it is hardly feasible to rely on a specific machine which would be responsible of the deployment. We made the most of the results obtained in [5] in which we have used a consensus algorithm to elect a manager that decides on the placement of a set of components. The consensus algorithm ensures that no contradictory decisions can be made in two different islands, e.g. the same component cannot be instantiated in two distinct islands.

The deployment descriptor contains the identity of the machines that are involved in the deployment. This requirement is necessary in order to define the notion of majority on which the consensus relies. However, when the deployment is triggered, some machines may not be connected. The first step of the deployment consists in broadcasting the architecture and deployment descriptors to at least one machine that belongs to the deployment target, which in turn broadcasts the descriptors to all the machines that are connected in the network. Each machine that receives these descriptors, creates the constraints described in the listing above depending on the deployment and architecture descriptors. Then a process is launched on each host. Locally, each machine maintains its own set of constraints (C1 to C6) and tries to make the deployment evolve until (a) solution(s) exist(s) for constraints C1, that is, some components can still be instantiated. The main steps of this process for the machine m_i and a component C that can be deployed on m_i are:

- For each resource constraint associated with C, a dedicated probe is launched (e.g. a probe to get the amount of free memory required by component C) in order to check if locally, all the required resources are available (C2). The observation of the resources is made periodically.
- If this is the case, that is, the component can be hosted locally, m_i sends its candidatures to all the machines involved in the deployment. This candidature indicates that m_i can host component C.
- Thus, m_i may receive several candidatures from others for the instantiation of C. When a candidature is received, m_i has to resolve a placement solution regarding to constraints C3. Depending on location constraints, a placement solution may require a sufficient number of candidatures
- Once a solution has been found by m_i, it tries to make it adopt by the consensus algorithm. If the consensus terminates, m_i updates the deployment descriptor with the new information of placement and broadcasts it to all the nodes that are currently connected.
- When a new descriptor is received, m_i updates the set C1 and C3 in order to take into account the placement decision made previously.
- m_i can then resolve some bindings towards newly instantiated (remote) components (C4) by sending a request to the machines hosting them. This is possible only if constraints C5 are still verified.
- When m_i receives a request of bindings, according to C6, it can accept or not this request and inform the sender of its answer.

– Depending on the answer, the definition domain that corresponds to the binding constraint (C4) is updated (removed from the constraint set if the binding is not possible or set to the remote host otherwise).

This process defines a propagative deployment driven by architectural and resources concerns. Since the observation of resources is made periodically, when a resource becomes available on a specific machine, this may yield the deployment to evolve. Similarly, when a machine enters the network (e.g. it is switch on), it announces its presence to the other nodes which will send it the current version of the architecture and deployment descriptors, making possible this newly connected machine to participate in the deployment evolution. In our current prototype, each machine maintains the list of connected hosts.

4 Related Work

Many ADLs provide capabilities to describe architecture patterns. Medvidovic and Taylor in [7] makes an overview of some existing ADLs offering such functionalities. Descriptions of architecture patterns with these ADLs make possible some reasoning about the modeled system, analysing its structure and evaluating its quality. At the best of our knowledge, only a few of these ADLs allow the enforcement of architecture patterns on an implementation deployed at runtime in a dynamic infrastructure. Some of the works targeting this goal are presented below. In addition, resource and location requirements are not well handled in all these languages in a homogeneous manner with architecture patterns like in R-ACL. If we would like to change an implementation technology (from Fractal to CORBA components, for example), R-ACL constraints can be easily transformed, as demonstrated in [12]. The solution adopted in this work which aims at transforming R-ACL constraints into runtime constraints makes also simpler the transformation of these new R-ACL constraints (in CORBA components ACL profile, for example).

We share similarities with researches on self-healing and self-organizing systems [6]. Indeed, the proposed approach here ressembles to the approach of [4, 10] in which the architecture of the system to deploy is not described in terms of component instances and their interconnections but rather by a set of constraints that define how components can be assembled. In both cases the running system is modelled by a graph. The main difference with our work is that reconfigurations of the systems are explicitly defined in a programmatic way while this is achieved automatically by the resolution of the constraints C_i in our work.

The work presented in [2] shares the same motivation to define high level deployment description with regard to constraints on the application assembly and on the resources the hosts of the target platform should meet. The authors present the Deladas language that allows the definition of a deployment goal in terms of architectural and location constraints. A constraint solver is used to generate a valid configuration of the placement of components and reconfiguration of the placement is possible when a constraint becomes inconsistent. This centralized approach does not consider resource requirements.

5 Conclusion & Ongoing Work

Deploying distributed systems in dynamic infrastructures remains a challenging task as resources and hosts availability cannot be predicted. In this paper we presented an approach which helps at assisting the deployment process with information about architecture patterns and resource requirements. This information is formally specified at design-time as constraints, written with a specific predicate language. These constraints allow the definition of complex component interaction and platform dependencies. In order to react on changes in the environment, these constraints are transformed and manipulated dynamically. By using these constraints, a propagative deployment is defined: components are instantiated as soon as needed resources become available and required hosts become connected while ensuring architecture consistency.

Our implementation is based on existing prototypes: ACE [11] for the description and the evaluation of ACL constraints, and a deployment manager based on Cream to maintain and solve runtime constraints. An evaluation of the behavior of our approach in a dynamic network is in progress.

References

1. E. Bruneton, C. T., M. Leclercq, V. Quéma, and S. Jean-Bernard. An open component model and its support in java. In *Proceedings of CBSE'04*, may 2004.
2. A. Dearle, G. N. C. Kirby, and A. J. McCarthy. A framework for constraint-based deployment and autonomic management of distributed applications. In *Proceedings of ICAC'04*, pages 300–301, 2004.
3. D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge Univ. Press, 2000.
4. I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proceedings of WOSS'02*, pages 33–38, 2002.
5. D. Hoareau and Y. Mahéo. Constraint-based deployment of distributed components in a dynamic network. In *Proceedings of ARCS 2006, LNCS, volume 3864*, pages 450–464, 2006.
6. J. Magee and J. Kramer. Self organising software architectures. In *Proceedings of FSE'96*, pages 35–38, 1996.
7. N. Medvidovic and N. R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1):70–93, 2000.
8. OMG. Corba components, v3.0, adpoted specification, document formal/2002-06-65. OMG Web Site: http://www.omg.org/docs/formal/02-06-65.pdf, June 2002.
9. OMG. Uml 2.0 ocl final adopted specification, document ptc/03-10-14. OMG Web Site: http://www.omg.org/docs/ptc/03-10-14.pdf, 2003.
10. B. R. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *In proceedings of SEKE'02*, pages 241–248, 2002.
11. C. Tibermacine, R. Fleurquin, and S. Sadou. Preserving architectural choices throughout the component-based software development process. In *Proceedings of WICSA'05*, pages 121–130, November 2005.
12. C. Tibermacine, R. Fleurquin, and S. Sadou. Simplifying transformations of architectural constraints. In *Proceedings of SAC'06, Track on Model Transformation*, April 2006.