Enforcing Architecture and Deployment Constraints of Distributed Component-based Software

Chouki Tibermacine[1], Didier Hoareau[1], and Reda Kadri[1,2]

[1] VALORIA, University of South Brittany, Vannes, France
{Didier.Hoareau,Chouki.Tibermacine}@univ-ubs.fr
[2] Alkante, Cesson Sévigné, France
r.kadri@alkante.com

In the component-based software development process, the formalisation of architectural choices makes possible to explicit quality attributes. When dealing with the deployment of such component-based software in dynamic networks, in which disconnections or machine failures can occur, preserving architectural choices becomes difficult to ensure, as current architecture-centric languages and their support mainly focus on steps prior to the deployment one. We present in this paper a family of languages that formalise not only architectural choices but deployment aspects as well, both as constraints. Then, we show how all of these constraints are reified in order to manage the deployment of a component-based software in this context of dynamic hosting platforms. The proposed solution defines an automatic deployment that ensures permanently, at run time, the preservation of architecture and deployment choices, and thus their corresponding quality attributes.

1 Introduction

Architectural choices should be preserved throughout the software lifecycle so that their associated quality attributes persist. For example, if we choose, at design-time, a particular architectural style like the pipe and filter [15], we should be able, at runtime, to enforce it so that maintainability and performance quality requirements can be ensured permanently.

In an MDE (Model-Driven Engineering) approach, we can define at architecture design-time an architecture description of a system with a given ADL, like Acme [4]. We can then transform this description into a component implementation in CORBA components (CCM) [10], for example. For a smooth transition, we can transit by a component diagram in UML 2 (or one of its profile, like CCM one), at component design-time. We showed in [16], how to formalize architectural choices at the different stages above using a family of constraint languages called ACL profiles: Acme ACL profile at architecture design stage, UML 2 ACL profile at component design stage and CCM ACL profile at component implementation stage. We also presented how these architectural choices (constraints) are preserved from one stage to another.

In this paper, we present how these choices can be preserved after the development has finished. We show how this can be achieved after the deployment of the component implementation in a distributed execution environment. Indeed, one of the characteristics of emerging distributed platforms is their dynamism. 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 inaccessible 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 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 choices. 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 initial 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 addition, we introduce in this paper the enrichment of architectural choices, during deployment-time, with constraints on resources and location. We show how we can use the same language to formalize this kind of constraints, and how we can check them at runtime. The proposed approach makes use of a transformation technique to evaluate ACL constraints. All architectural choices together with resource and location constraints are transformed into reified runtime constraints to be evaluated.

In the next section we present briefly how we can formalize architecture choices using a constraint language, and we illustrate this formalization by a short example of a client/server architectural style. In addition we show how to use this same language to describe resource and location requirements at component deployment stage. We present in section 3, the deployment process and the resolution mechanisms of these constrained component-based software in dynamic infrastructures. In section 4 implementation details and experiment results are given. Before concluding and highlighting the perspectives, we present some related work in section 5.

2 Formalizing Architectural Choices during Development

In order to make explicit architectural choices, like the use of a particular architecture style or the enforcement of general architecture invariants, we proposed

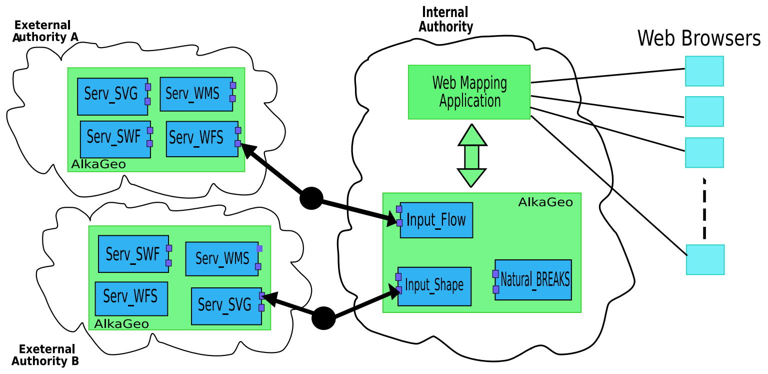

Fig. 1. Client/Server architecture of a Web mapping system

in [16] a constraint language named ACL (Architecture Constraint Language). Architectural choices are thus formalized 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 [11], 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[3], for UML 2, for OMG's CORBA Components (CCM) or the profile for ObjectWeb's Fractal [1].

To illustrate our work, we briefly describe the development process of a component-based software we developed, from the architecture design stage to the deployment stage. We chose xAcme to illustrate the architecture design stage and the Fractal component model for the implementation stage.

2.1 Architectural Choices at Architecture Design Stage

As an answer to a request from a local community in Brittany (France), we developed a component-based software, called AlkaGeo. This software generates geographic information flow which is used by a Web Mapping Application (WMA). When using this application, our customer can access Web GIS data and maps, like land maps, through their browsers. This WMA is deployed in application servers of our provider (Internal Authority, in Figure 1).

[3] xAcme is an XML extension of Acme ADL.

The overall architecture of AlkaGeo is organized according to the client/server style. In this system we have two instances of this style. The first occurrence of this style can be seen in Figure 1 between the components (Input_Flow) asking for maps and data, in two different formats SVG and SWF, from server components (Serv_SVG and Serv_SWF). The second instance of the style is defined between clients (Input_Flow) requesting maps and data in the GML format from server components (Serv_WMS and Serv_WFS)[4]. AlkaGeo is deployed on different server providers (External Authorities), which have different resources and configuration, to which we do not have access. For the sake of brevity, we illustrate in this work just the GML flows service implemented by the Serv_WMS and Serv_WFS components.

The client/server style is characterized by the following constraints:

– There is no direct communication between Input_Flow components,
– Serv_WFS and Serv_WMS can accept requests from at most 40 different Input_Flow components,
– Input_Flow components can use at most one Serv_WFS component or one Serv_WMS component.

These three constraints can be described using ACL profile for xAcme as follows:

1.
```
context ClientServer:ComponentInstance inv:
ClientServer.subArchitecture.archInstance.linkInstance->select(l|
l.endPoint->forAll(p1,p2|p1.anchorOnInterface.componentInstance
.id = 'Input_Flow' and p2.anchorOnInterface.componentInstance
.id <> 'Input_Flow'))
```

   This constraint states that for all link instances between architecture instances, there should be no link which binds two components which are identified by Input_Flow.

2.
```
context ClientServer:ComponentInstance inv:
ClientServer.subArchitecture.archInstance.componentInstance
->forAll(c| ((c.id = 'Serv_WFS') or (c.id = 'Serv_WMS')) and
(c.linkInstance->select(l|l.componentInstance
.id = 'Input_Flow'))->size() <= 40)
```

   The constraint above stipulates that component instances with the identifier Serv_WFS and Serv_WMS should have at most 40 links with component instances with the identifier Input_Flow.

3.
```
context ClientServer:ComponentInstance inv:
ClientServer.subArchitecture.archInstance.linkInstance
->forAll(l|l.endPoint->select(l|l.endPoint->forAll(p1,p2|
(p1.anchorOnInterface.componentInstance.id = 'Input_Flow')
and ((p2..anchorOnInterface.componentInstance.id = 'Serv_WFS')
or (p2..anchorOnInterface.componentInstance.id = 'Serv_WMS')))
```

   The last constraint enforces the existence of at most one link between the component instance with the identifier Input_Flow and one of the two component instances identified by Serv_WFS and Serv_WMS.

[4] WMS and WFS are two standards of the Open Geospatial Consortium: http://www.opengeospatial.org/

ACL profile for xAcme is composed of CCL and a MOF metamodel of xArch. An xArch architecture instance is composed of a set of component instances, connector instances, link instances and logical groups of the previous architectural elements. Component or connector instances define a set of interface instances and optionally a sub-architecture for a hierarchical description. The sub-architecture defines a set of architecture instances and a list of mappings between inner and outer interface instances. Link instances bind two end points, each one references an interface instance. As we can see, the constraints above navigate in this xArch metamodel.

2.2 Architectural Choices at Component Design Stage

Before implementing our software, we decided to establish an intermediate UML model for a smooth transition. Indeed, recent experiments [13] showed also that some ADLs and the UML can be used in a complementary fashion, in order to make better analysis of software architectures. The constraints formalizing the client/server style can be described, at this stage, using the ACL profile for UML 2. The first constraint is expressed as follows:

```
context ClientServer:Component inv:
ClientServer.connector.end.role->forAll(r1,r2|(r1->oclAsType(Port)
.encapsulatedClassifier->oclAsType(Class)->oclAsType(Component)
.name = 'Input_Flow') and (r2->oclAsType(Port).encapsulatedClassifier
->oclAsType(Class)->oclAsType(Component).name <> 'Input_Flow'))
```

This constraint navigates in the UML 2 component metamodel. At the differences of the previous constraint, it manipulates connectors, roles, components and ports. The constraint above and the first constraint expressed in the previous subsection have the same semantics in the context where they are applied (on an xAcme architecture description and on a UML 2 component model).

In order to evaluate constraints, we use an intermediate ACL profile to which all architectural constraints specified in the different profiles are transformed. At a given stage of the development process, architecture choice preservation is achieved by the transformation of constraints specified in all upstream stages in this intermediate ACL profile to be evaluated.

2.3 Architectural Choices at Component Implementation Stage

Suppose that the system modeled above has been implemented in a component technology, like Fractal. The three constraints of the previous client/server style can be described at this development stage using ACL profile for Fractal. In the listing below, we illustrate the first constraint expressed in this profile:

```
context ClientServer:CompositeComponent inv:
ClientServer.binding->forAll(b|b.client.component.name="Input_Flow"
and b.server.component.name <> "Input_Flow")
```

This constraint navigates in the MOF metamodel of Fractal component model which is presented in Figure 2. This metamodel abstracts components, which can be composite or primitive. Components can have interfaces of several

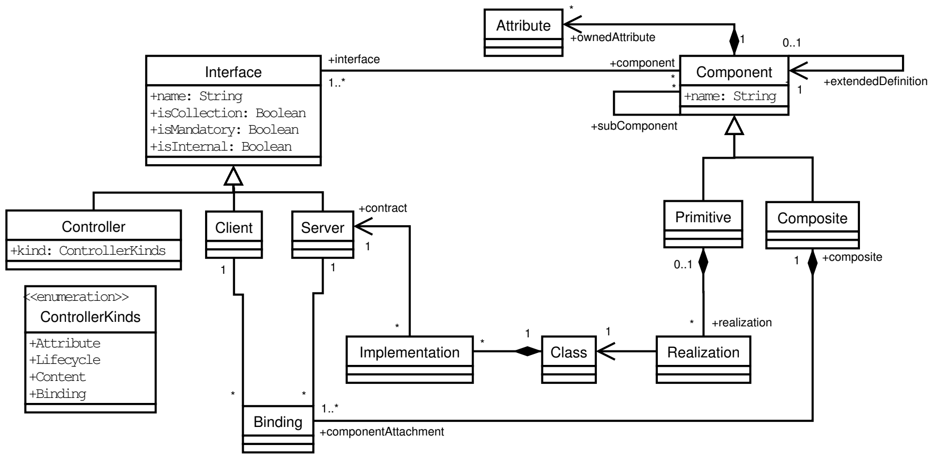

Fig. 2. The MOF metamodel of Fractal component model

types. Server interfaces are interfaces that specify provided functionalities. Client interfaces define required operations. 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 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).

2.4 Resource and Location Requirements at Deployment Stage

In addition to these architecture constraints, the deployment of each component is governed by some resource and location requirements. Indeed, before deployment, we are unlikely to know what are the machines that are involved in the deployment and thus where to deploy each component. However, one can define for each component what are its requirements in terms of resources, that is, the characteristics of the machines that will host the component. For example, a Serv_SVG must be hosted by a machine that has at least 512 MB of free memory, a CPU scale greater than 1 GHz and is connected to the network by an interface with a bandwidth of at least 512 Kb/s. With regard to Input_Flows, each instance must be hosted by a machine that belongs to the Internal Authority provider.

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. The resource constraints introduced above are described in R-ACL as follows:

1. Free memory >= 512 MB:

```
context Serv_SVG:Component inv:
Serv_SVG.resource->oclAsType(Memory).free >= 512
```

2. CPU scale > 1 GHz (1000 MHz):

```
context Serv_SVG:Component inv:
Serv_SVG.resource->oclAsType(CPU).processors
->select(cpu:CPU_Model|cpu.speed>1000)->size()>=1
```

3. Network interface bandwidth >= 512 Kb/s:

```
context Serv_SVG:Component inv:
Serv_SVG.resource->oclAsType(NetworkInterface).tx >= 512
```

4. Each instance of the component Input_Flow must be hosted by an internal authority machine:

```
context Input_Flow:Component inv:
Input_Flow.location->forAll(h:Host|h
.group = 'Internal Authority')
```

As discussed above these constraints navigate in the resources metamodel, but have as a scope only one specific architectural element, which is the component Server_SVG or Input_Flow. This element is of type Component which is a sub-meta-class of the meta-class ArchitecturalElement. This meta-class is the ancestor of all meta-classes in the Fractal metamodel[5].

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 Serv_SVGs 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.name='Serv_SVG'
and c2.name='Serv_SVG' and c1.location.id <> c2.location.id)
```

The different categories of constraints are saved in XML documents. There is a style descriptor which contains the constraints formalizing an architecture style (the first category of constraints) and a deployment descriptor which embeds resource and location constraints (the second and the third category of constraints). These descriptors are used while deploying the system, as described in the next section.

3 Preserving Architectural Choices at Runtime

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 architecture choices

[5] This is omitted from Figure 2 for the purpose of clarity.

and resource specifications. Then, for the purpose of clarity, we detail first the deployment process in a non-partitioned network—this will allow us to focus on the dynamic resolution of constraints—then we take into account fragmentation within the environment.

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 style 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 for every instance. 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. As it is detailed below, these runtime constraints are suited when considering reaction mechanisms to changes that can occur in the environment. The reified constraints are generated 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 Prolog IV [12] can then be used to find one or several solutions. On the one hand the use of dynamic constraints makes it possible to preserve architectural choices at runtime, on the other hand reified constraints allow detecting and reacting to changes that can occur with the environment. By identifying these different changes we will explicit the constraints that have to be reified and that will guarantee the preservation of the architecture's consistency all along its execution.

In the kind of network we qualify as dynamic, crashes (e.g. failure of machines, components) may happen and partitions may exist. In both cases, some components that were in use regarding other components can become unavailable. When dealing with a crash, if some repair mechanisms have been defined, these components can or must be redeployed. However because of the existence of islands, it is crucial to control the instantiation mechanism (and thus the number of instances). Indeed, the strategy consisting in redeploying a component each time this latter fails is not suited as the number of instances will not be consistent even if it is the case in each island. In order to overcome the instantiation of components in a dynamic network, we introduce a first type of constraints, named C_1:

C1 These constraints specify the number of instances allowed for each component. By fixing the minimum and the maximum of instances allowed of a component it is possible to control its instantiation which can be initiated due to the dynamism of the network. When a new resource, required by a component, becomes available, its instantiation is conceivable. In the same way, when a component becomes faulty or becomes out of reach, one may consider its substitution by a new instance. Due to partitions within the network, it is mandatory not only to have such a constraint but to maintain its consistency as well: the information about the current number of instances is a global one, and thus must be the same within each island.

In a dynamic network resources on machines may change in such a way that a required resource that was unavailable when the deployment was triggered, may become available later. Moreover, because of the mobility of the devices that compose the network we target, some machines that were out of reach until now may become accessible, inducing the availability of new (required) resources[6]. In order to take into account changes of resources and hosts, we introduced constraints C_2 and C_3:

C2 It is possible with R-ACL to define components' needs in terms of software and hardware resources. In order to react on resources'changes, resource constraints are reified and form constraints C_2;

C3 In the same way, location constraints have to be reifed to take into account hosts mobility. When dealing with a constraint such specifying that components $\mathsf{Serv\_SVG}_1$ and $\mathsf{Serv\_SVG}_2$ must reside on two distinct hosts, a deployment may initially not be possible due to the absence of one or several hosts. A solution can however be found as soon as the number of connected (and reachable) hosts is sufficient. Constraints C_3 correspond to the reification of location constraints.

The constraints presented above allow to react on changes of the environment, that is, the fluctuation of resources and the mobility and volatility of hosts, while controlling the number of instances of the components. When a component instance is created or withdrawn, the architecture of the application, i.e. the assembly of the components, has to be reconfigured : indeed, when a component is created, some bindings have to be added towards this component, and if the latter requires others, bindings to these components have also to be made. When dealing with the removal of a component, bindings towards and from this component have to be suppressed. The addition and suppression of bindings on any architecture must be done regarding the architectural constraints defined at design-time. For example, the client / server style of AlkaGeo specifies that at most 40 component Input_Flow can be bound to component Serv_WFS. Thus, we introduce three more constraints that are reified and that preserve the architectural constraints during bindings reconfiguration.

C4 When a component is instantiated in consequence of the availability of new resources or when a remote component becomes accessible, it is mandatory to add it into the architecture (i.e. to set up bindings) if the style descriptor specifies the interconnection of this component with others. Constraints of type C4 are the reification of information specifying a binding between two components or two types of components.

The previous constraints make it possible to detect that a binding between two components can be made once the style descriptor specifies such a binding

[6] Besides, a resource used by a component may become unavailable (e.g. the amount of free memory).

and that the two components are reachable from each other. Even if a binding can be made, some other aspects can prevent this creation. For example, the AlkaGeo application defines a client / server style which limits component Input_Flow to use at most one component Serv_WFS, and that every component Serv_WFS can only be used by at most 40 components Input_Flow. It is thus necessary, before creating a binding between a component Input_Flow and a component Serv_WFS to check that the number of connections respects the architectural choices. Constraints C5 and C6 reify these constraints:

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. These sets are sufficient to generate a valid configuration regarding to an architectural style. 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.

3.2 Deployment Process: A Centralized Evolution

We will consider first a network in which no fragmentation into islands is possible (this assumption will not be considered in the next subsection). Further, we make the following assumptions: there is a dedicated machine, called DeployManager on which we can rely in order to maintain up-to-date the ids of the machines that are connected. When the deployment is triggered, some machines may not be connected. Besides, a machine that enters the network is detected by the DeployManager.

When the deployment is launched, style and deployment descriptors are sent to the DeployManager, 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 style 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 (or multiple) solution(s) exist(s) for constraints C1, that is, some components can still be instantiated. The main steps of this process for a component C that can be deployed in a machine m_i are the followings:

– For each resource constraint associated with C, a dedicated probe is launched (e.g. a probe to get the amount of 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 the DeployManager. This candidature indicates that m_i can host component C.
– The latter may receive several candidatures from other machines for the instantiation of C. The DeployManager 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, the DeployManager 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 by the DeployManager.
– 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 refuse 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 requirements. 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, the DeployManager sends the current version of the style and deployment descriptors to this machine, making possible this newly connected machine to participate in the deployment evolution.

3.3 Deployment Evolution in a Partitioned Network

The deployment described above relies on a dedicated machine—the DeployManager—that orchestrates the evolution of the deployment regarding to the resolution of the location constraints. In front of islands, that is, the fragmentation of the network, the uniqueness of such a manager raises the problem of the propagative deployment in islands where no manager exists. We have addressed this aspect by considering the management of several managers. The main difficulties here are twofolds: first, how can we guarantee the architecture consistency if several managers make decisions independently to each other (e.g. we have to avoid the instantiation of the same component in two distinct islands) ? Secondly, the management of multiple managers have to be faced with when two islands *merge*.

We have decided to use the results obtained in [6] in which we have defined a consensus algorithm to elect such a manager in networks where partitions can occur. This algorithm is based on a common view of the different machines to make a decision about the identity of an approved manager. Thus, the resolution of location constraints can be made in islands composed of a majority of machines. The consensus algorithm ensures that no contradictory decisions can be made in two different islands and that the latest version of the style and deployment descriptor exists in every island.

Unlike the centralized version of the propagative deployment, the deployment presented in partitioned network requires that the ids (thus the number) of the machines that will be involved in the deployment, be known. Indeed the used algorithm depends on a majority of connected machines, in order to terminate.

4 Implementation status and results

In order to validate our proposals, we enhanced and reused some existing prototype tools. The first tool is ACE (Architecture Constraint Evaluator). ACE is composed of an editor for ACL constraints. This editor assists developers to write their constraints by proposing the different navigation alternatives in the used metamodel (resources and location metamodel or architecture metamodel). After specifying these constraints, ACE makes some well-formedness checking and compiles them in order to generate the corresponding runtime constraints. This transformation process is performed starting from a Java implementation of the abstract syntax tree of the different constraints.

The constraints that are solved dynamically have been implemented with Cream[7]. Cream is a Java library for writing and solving constraint satisfaction problems or optimisation problems on integers. Every constraint generated from an R-ACL's one defines a relation on a variable taking its value in a finite domain. For the location constraints, the definition domain of each variable is not known before the deployment but is increased each time a candidature is received.

The deployment that has been presented in this paper relies on the discovery of the resources required by the components. For that, we used DRAJE (Distributed Resource-Aware Java Environment) [7], an extensible Java-based middleware developed in our team. Thus, hardware resources (e.g. processor, memory, network interface...) or software resources (e. g. process, socket, thread, directory...), can be modelled and observed in a homogeneous way. For every resource constraint of the deployment descriptor, a resource in DRAJE is created and a periodic observation is launched.

The performance of the deployment process depends on changes imposed by the execution environment such as resources availability and host connectivity. But, the propagative deployment requires the DeployManager to solve first a solution placement before the instantiation can go along. Hence, we have measured the impact of this computation. The preliminary results of this experiment showed that the time to obtain a placement solution (when all conditions are met) remains acceptable (less than 10 mili-seconds to deploy 50 Serv_SVG components) and corresponds to the complexity of the AllDiff constraint (i.e. each Serv_SVG must be hosted on a distinct machine) which is $O(n^2)$.

5 Related Work

Many ADLs provide capabilities to describe architecture choices. Medvidovic and Taylor in [8] make an overview of some existing ADLs offering capabilities to describe architectural styles and constraints in general. The description of architecture styles with these ADLs makes possible some reasoning about the modeled system, analysing its structure and evaluating its quality. The difference between the work presented here and such ADLs is twofolds:

[7] http://kurt.scitec.kobe-u.ac.jp/∼shuji/cream/

– First, design-level and deployment-level constraints are described in a homogeneous way in our approach. Indeed, the same language (ACL) is used throughout the software life-cycle to describe them. The majority of ADLs deals only with one kind of these constraints. Some ADLs focus on architectural style description, like Aesop [3]. Others, deal with deployment requirements specification, like in [6]. Even if an ADL deals with the two kinds of constraints at the same time, there is no means to describe them at different stages of the development process. In these ADLs, architecture design and deployment requirements should be addressed together and language constructs that are used to specify them are mixed. The approach we propose here targets the separation of concerns by providing a single constraint language, with many profiles; each profile can be used to deal with a particular concern (design choice formalization or deployment requirement description).
– Second, the approach proposed here is implementation technology-independent. An easy migration can be performed from one implementation technology to another, as demonstrated in [17]. However in existing works, constraint languages are tightly coupled with ADLs, and constraints are parts of architecture or component descriptions. This makes difficult migration between technologies, because whole architecture descriptions should be translated.

We share similarities with researches on self-healing and self-organizing systems. Indeed, in the approaches presented in [5, 14], a system architecture 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 (C1 to C6) in our approach.

In [9], the authors present an approach to deploy software components in resource constrained environments. The deployment process is initiated by the Continuous Analysis component which maintains up-to-date the current topology of the running application. This component is responsible of initiating the necessary operations to deploy a part of the architecture if there is a difference between the current and desired configuration. The deployment of a given component is performed starting from an architecture description specified with an ADL called PitM ADL, which is interpreted by the Prism architecture middleware. Besides this centralized version, the authors specified a distributed ownership of the deployment process in which several Continuous Analysis components are responsible of the deployment of a local subsystem. This distributed process differs from ours as it relies on the division of the system into subsystems which cannot be done *a priori* in a network with evolving topology; such dynamic networks are not considered by the authors.

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 placements of components and reconfiguration of the placement is possible when a constraint becomes inconsistent. This centralized approach requires, contrary to ours, a full knowledge of the identity of the different hosts that may participate in the deployment. Moreover, the current version of Deladas does not consider resource requirements.

6 Conclusion & Future Work

Preserving architectural choices throughout the development process of a software is an important aspect. Indeed, in order to implement a software that complies with the initial requirements, architectural choices should be formalized at all stages. In addition, at a given stage, architectural choices defined in upstream stages should be preserved. This makes possible a traceability of quality attributes implemented by these choices. After the implementation of this system comes its deployment. Another aspect is important in the life-cycle of the developed software. It is related to the preservation of architectural choices after its deployment (during its execution). Indeed, this makes the system benefit from the quality attributes, associated to these choices, which can be dynamically observed, like performance or reliability. In the example introduced in section 2, the client/server style is formalized and enforced dynamically, in order to benefit from the dynamic quality attributes guaranteed by this style (like, scalability and interoperability).

In this paper, we presented an approach to formalize, as constraints, architecture choices made throughout a component-based software life-cycle. We illustrated how we can use the same formalization language (ACL) to describe resource and location requirements that appear at deployment stage. We showed how these constraints are checked while deploying the implemented system in a dynamic infrastructure. Indeed, in this kind of platforms the availability of resources and hosts cannot be predicted. Faced with the environment evolution (disconnection and reconnection of nodes), we presented a deployment process that checks permanently the constraints to enforce architecture choices with respect to deployment requirements. The constraints that are checked dynamically are obtained after transforming ACL static constraints into runtime (CSP) ones.

We are working now on defining architecture patterns as libraries which are automatically transformed into their equivalents at runtime. This will, as we think best, make easier architecture description, more specifically architectural style formalization, and will simplify considerably the deployment of its implementation according to the proposed approach.

Even if not considered in this article, the management of network failures[8] is one of our current work. The main difficult aspect resides in the automation of the re-deployment regarding the constraints resolution mechanism.

[8] In the case of a partitioned network, one can notice that the distinction between the failure of a machine and its inaccessibility is a hard problem.

References

1. E. Bruneton, C. Thierry, M. Leclercq, V. Quéma, and S. Jean-Bernard. An open component model and its support in java. In *Proceedings of CBSE'04*, Edinburgh, Scotland, 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. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of FSE'94*, pages 175–188, New Orleans, Louisiana, USA, 1994.
4. D. e. a. Garlan. Acme: Architectural description of component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.
5. I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *Proceedings of WOSS'02*, pages 33–38, Charleston, South Carolina, USA, 2002.
6. D. Hoareau and Y. Mahéo. Constraint-Based Deployment of Distributed Components in a Dynamic Network. In *Proceedings of ARCS'06*, volume 3864 of *LNCS*, pages 450–464, Frankfurt/Main, Germany, March 2006. Springer Verlag.
7. Y. Mahéo, F. Guidec, and L. Courtrai. A Java Middleware Platform for Resource-Aware Distributed Applications. In *Proceedings of ISPDC'03*, pages 96–103, Ljubljana, Slovenia, October 2003. IEEE CS.
8. N. Medvidovic and N. R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE TSE*, 26(1):70–93, 2000.
9. M. Mikic-Rakic and N. Medvidovic. Architecture-level support for software component deployment in resource constrained environment. In *Proceedings of the 1st International IFIP/ACM Conference on Component Deployment (CD'02)*, pages 31–46, Berlin, Germany, 2002.
10. OMG. Corba components, v3.0, adpoted specification, document formal/2002-06-65. Object Management Group Web Site: http://www.omg.org/docs/formal/02-06-65.pdf, June 2002.
11. OMG. Uml 2.0 ocl final adopted specification, document ptc/03-10-14. Object Management Group Web Site: http://www.omg.org/docs/ptc/03-10-14.pdf, 2003.
12. I. Prolog. constraints inside, 1996. *Prolog IV reference manual*, 1996.
13. R. Roshandel, B. Schmerl, N. Medvidovic, D. Garlan, and D. Zhang. Understanding tradeoffs among different architectural modeling approaches. In *Proceedings of WICSA'04*, pages 47–56, June 2004.
14. B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of SEKE'02*, pages 241–248, Ischia, Italy, 2002.
15. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
16. C. Tibermacine, R. Fleurquin, and S. Sadou. Preserving architectural choices throughout the component-based software development process. In *Proceedings of WICSA'05*, Pittsburgh, Pennsylvania, USA, November 2005.
17. C. Tibermacine, R. Fleurquin, and S. Sadou. Simplifying transformations of architectural constraints. In *Proceedings of SAC'06*, Dijon, France, April 2006.