A Framework for Autonomic Software Deployment of Multiscale Systems

Raja BOUJBEL
Université de Toulouse
UPS - IRIT
118 Route de Narbonne
F-31062 Toulouse, France
Raja.Boujbel@irit.fr

Sébastien LERICHE
Université de Toulouse
ENAC
7 Avenue Édouard Belin
F-31055 Toulouse, France
Sebastien.Leriche@enac.fr

Jean-Paul ARCANGELI
Université de Toulouse
UPS - IRIT
118 Route de Narbonne
F-31062 Toulouse, France
Jean-Paul.Arcangeli@irit.fr

Abstract—**Automated deployment of software systems in pervasive and open environments is an open issue. There, the topology of target hosts is not always known at design time due either to unforeseen hardware limitations or failures (network links, hosts, etc.) or to device arrival and disappearance. Rather than manually building and executing a static deployment plan, as it is usually done, our approach promotes the specification of deployment properties (requirements and constraints), then their handling by a middleware for autonomic deployment. This paper presents MuScADeL, a new domain-specific language designed to support the expression of properties related to multiscale and autonomic software deployment. It also presents a chain of software tools that participate in the deployment process and are part of the autonomic deployment middleware, including a system of probes for the monitoring of the host machines and a compiler of multiscale deployment properties.**

Keywords—**Software deployment, multiscale distributed systems, domain-specific language, autonomic computing, constraint satisfaction problem.**

I. Introduction

Pervasive computing, on the one hand, and cloud computing, on the other hand, are central topics in several recent research studies. Contributions in both domains have reached a good level of maturity. Nowadays, new research works have identified the need to make pervasive and cloud computing systems collaborate, so as to build systems which are distributed over several scales, called "multiscale" systems. In multiscale systems, decentralization, autonomy and adaptiveness are essential features.

In this context, our work focuses on software deployment and our goal is to develop a framework for supporting the deployment of multiscale applications. Deployment aims at making and keeping software systems available for use, in a situation of mobility, openness and variability of the quality of the resources. Deployment strategies should take into account the multiscale aspects like geography, network, device, and user, as well as non functional properties such as efficiency and privacy.

In this paper, we describe a Domain-Specific Language (DSL) dedicated to multiscale and autonomic software deployment, named MuScADeL (***Mu**lti**Sc**ale **A**utonomic **De**ployment **L**anguage*) [1], then we present how the deployment plan can be computed from the MuScADeL specification.

In the rest of this section, the novel concept of multiscale system and the basics of software deployment are introduced, then the problem of multiscale software deployment is analyzed, and the requirement of a DSL is expressed. Finally, Section I-E presents the plan of the article.

A. Multiscale distributed systems

The term "multiscale system" is present in several recent research papers [2], [3], [4]: in these works, authors consider to make collaborate very small systems (objects from the Internet of Things paradigm as, for example, swarms of tiny sensors with very low computing capabilities) with very big systems (such as those found in cloud computing). They agree that new issues arise, mainly those related to huge heterogeneity.

The INCOME project [5] aims at designing software solutions for context management in multiscale systems, that is to say not only in ambient networks, but also in the Internet of Things and the Cloud, able to operate at different scales and to deal with the passage from a scale to another one. Context management is a complex service in charge of the gathering, the management (processing and filtering), and the presentation of context data to applications, which realization is distributed on the different devices which compose the system. So, context managers are open multiscale applications, and we are interested in their deployment.

In [6], Rottenberg *et al.* argue that the multiscale nature of a distributed system should be analyzed independently in several specific viewpoints such as geography, network, device, data, user, etc. Thus, a distributed system can be described as multiscale when, given a viewpoint, for at least one dimension of this viewpoint, the elements of its projection onto this dimension are associated with different scales. Fig. 1, extracted from [7], shows an example of scales in the "Processing power" dimension in the "Device" viewpoint. The dimension "Processing power" is composed by several scales: kilo scale, giga scale, and peta scale. A family of device can be contained in one scale, as personal devices in the kilo

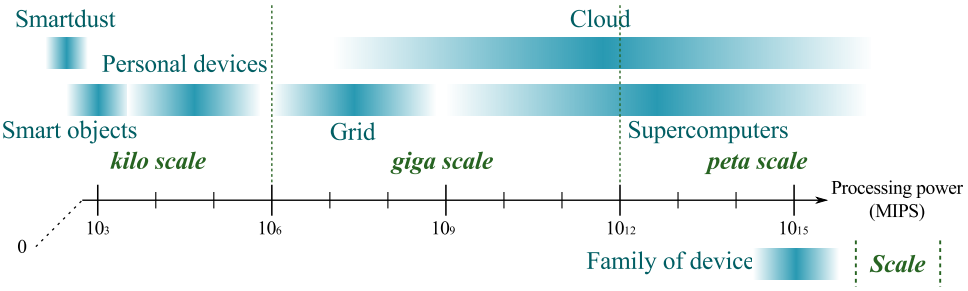

Fig. 1: Scales in the "Device Processing power" dimension.

scale, or in more than one scale, as supercomputers in giga and peta scales.

However, the concept of "multiscale system" is not actually mature. The construction of future multiscale distributed systems will necessitate new kinds of languages, middleware and patterns, allowing to take in consideration the multiscale aspects of the systems.

B. Software deployment

Software deployment is a post-production process which consists in making software available for use and then keeping it operational. It is a complex process that includes a number of inter-related activities such as installation of the software into its environment (transfer and configuration), activation, update, reconfiguration, deactivation and deinstallation [8]. Fig. 2 represents the sequence of the activities. Software release and software retire are carried out on the "producer site", while the other activities are carried out on the "deployment site", some of them at application runtime.

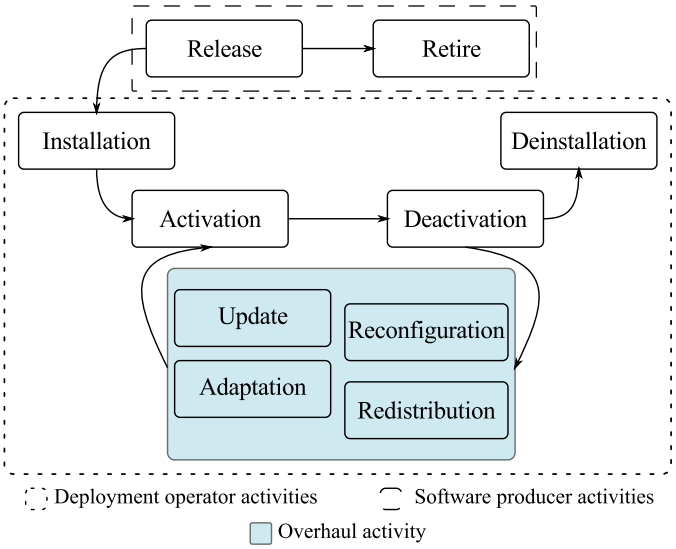

Fig. 2: Software deployment life cycle.

Deployment design is handled by an engineer called "deployment designer". He has to gather information not only about the software system to deploy and the properties of each of its components but also about the distributed organization of the software at runtime. Designing deployment may consist in expressing requirements and constraints. For instance, the deployment designer may express that a particular software component should be installed on some specific devices or on any device, even on incoming ones in case of dynamic systems, while satisfying a set of properties. As a concrete example, consider a software component C which should be deployed on each smartphone which runs Android, has the GPS function active, and is connected by WiFi.

A deployment plan is a mapping between a software system and the deployment domain, increased by data for configuration (and about dependencies). The deployment domain is the set of networked machines or devices which hosts the components of the deployed software system. The ultimate purpose of deployment design is to produce a deployment plan which complies with the expressed properties. Usually, this task is undertaken by a human actor.

At runtime, software must be deployed on the domain according to the deployment plan, this task being possibly undertaken or controlled by an operator called "deployment operator". Automatization of deployment aims at avoiding (or limiting) human handling in the management of deployment.

Fig. 3 shows the timeline of deployment.

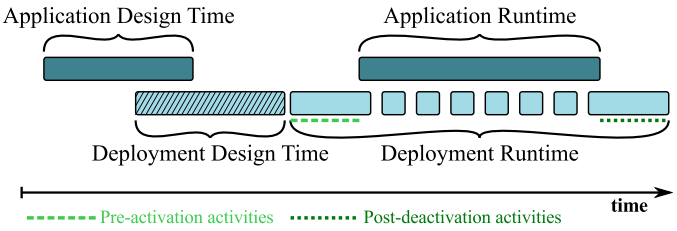

Fig. 3: Software deployment timeline.

C. Multiscale software deployment

In this work, we focus on deployment design, and particularly on the ways for a deployment designer to express multiscale deployment properties.

Software deployment in large-scale and open distributed systems (such as ubiquitous, mobile or peer-to-

peer systems) is still an open issue [9]. There, existing tools for software deployment are reaching their limits: they use techniques that do not suit the complexity of the issues encountered in such infrastructures. Indeed, they are only valid within fixed network topology and do not take into account neither host and network variations of quality of service nor failures of machines or links which are typical of these environments.

Moreover, users of the deployment tools are required to manage manually the deployment activities, which needs a significant human involvement, possibly out of reach of concerned end-users (for example, in case of personal devices like smartphones): for large distributed component-based applications with many constraints and requirements, it is too hard and complicated to accomplish the deployment process manually. Consequently, there is a need for new infrastructures and techniques that automate the deployment process and allow a dynamic reconfiguration of software systems with few or without human intervention.

Additionally, in our opinion, decentralization, openness and dynamics (mobility, variations of resources availability and quality, disconnections, failures) are in favor of autonomy: the autonomic computing approach [10], where the system self-manages some properties (self-configuration, self-healing), may support solutions which satisfy the requirements of distributed multiscale software systems deployment. This idea lead us to "autonomic software deployment" [9].

Instead of directly expressing a statically defined deployment plan, we propose to express *properties*: deployment *requirements* and component *constraints* from which the deployment plan can be computed. In this paper, we focus on the expression of the properties, and on the construction of the plan.

So, in order to build the plan, and moreover to allow management of deployment at runtime, data about the domain must be collected. Thus, a system of probes should run and collect data ranging from the domain properties such as free RAM to more abstract ones related to multiscale (viewpoints, dimensions, and scales). Relations between probes and properties can be made explicit at the same level as the deployment properties in order to allow the specification of the system of probes at the deployment design time.

D. Towards a DSL for autonomic software deployment of multiscale systems

In this ongoing work, our aim is to provide a solution for the expression of the deployment design, concerning in particular the scales and other significant properties of multiscale software systems (see below an example in Section III).

Deployment is a specific operation on software. Its design requires particular skills. Thus, we think that the deployment designer could benefit from a dedicated language when stating the properties. So, we propose a DSL dedicated to the description of deployment constraints and requirements. DSLs present several advantages: they use idioms and abstractions of the targeted domain, so they can be used by domain experts; they are light, so easy to maintain, portable, and reusable; they are most often well documented, coherent and reliable, and optimized for the targeted domain [11], [12], [13].

E. Plan of the article

The rest of the paper is structured as follows. Section II discusses related work on DSL-based software deployment. Section III provides an example of deployment of a multiscale software system. The DSL MuScADeL is presented in Section IV using the example presented in Section III. Section V introduces the software elements that complement MuScADeL in order to compute a deployment plan. Section VI presents the bootstrap of the deployment management system, its architecture, and the interface used by the deployment operator. Section VII explains how deployment properties are transformed and formalized. Section VIII presents our constraint solving library and its use through the MuScADeL specification presented in Section IV. Section IX concludes and discusses some future works.

II. Related work on DSL-based software deployment

The need for automation in software deployment has given to this activity a special attention both in academia and in industry. There are a large number of tools, procedures, techniques, and papers addressing different aspects of the software deployment process from different perspectives.

Existing deployment platforms propose several formalisms to express deployment constraints, software dependencies, and hardware preferences of software to deploy. Usually, the formalisms include architecture description languages (ADL), deployment descriptors (like XML descriptor deployment), and dedicated languages (DSL). In this section, we overview some works on software deployment that propose the use of a DSL.

Fractal Deployment Framework (FDF) [14] is a component based software framework to facilitate the deployment of distributed applications on networked systems. FDF is composed of a high-level deployment description language, a library of deployment components, and a set of end-user tools. The high level FDF deployment description language allows end-users to describe their deployment configurations (the list of software to deploy and the target hosts). Finally, FDF provides a graphical user interface allowing end-users to load their deployment configurations, execute and manage them. The deployment unit is an archive that contains the software binary and the deployment descriptor. The

main limitation of this tool is the static and manual attributes of the deployment. Although the static deployment plan is eligible in a stable environment like Grid, this deployment is not usable in an environment characterized by a dynamic network topology such as ubiquitous environments. Another limitation is that in runtime this tool does not provide mechanisms for dynamic reconfiguration which allows the treatment of the hosts and the network failures.

Dearle *et al.* [15], [16] present a framework for autonomic management of deployment and configuration of distributed applications. To facilitate the work of the deployment designer, they define a DSL, Deladas. Using it, a set of available resources and a set constraints are specified. These definitions permit to generate an applicable deployment plan. The constraint-based approach avoids the deployment designer specifying precisely the location of each component, and then rewriting all the plan in case of problems with a resource. Deladas does not allow to express multiscale properties and constraints. Openness is neither taken into account, the set of hosts is statically defined in a file by the deployment designer. Deployment is still autonomic: at runtime, when the deployment middleware detects a constraint violation (dependencies between components), it tries to solve it by a local adaptation. The new deployment plan is computed by a centralized management component called MADME.

Matougui *et al.* [9] present a middleware framework designed to reduce the human cost for setting up software deployment and to deal with failure-prone and change-prone environments. This is achieved by the use of a high-level constraint-based language and an autonomic agent-based system for establishing and maintaining software deployment. In the DSL called j-ASD, some expressions dedicated to deal with autonomic issues are proposed. But they target large-scale or dynamic environments such as grids or P2P systems, only within the same network scale.

Sledviewsky *et al.* [17] present an approach that incorporates DSL for software development and deployment on the cloud. Firstly, the developer defines a DSL in order to describe a model of the application with it. Secondly, the application is described using the DSL, then it is translated into specific code and automatically deployed on the Cloud. This approach is specific to the deployment of a Web application on the cloud. It highlights the need to facilitate the work of the deployment designer, and that using DSL is a solution for that.

Recent works around software deployment start taking into account constraints of quality of service. For example, Malek *et al.* [18] present a framework (tools and formalism) aiming at determining a ”best” deployment plan regarding several constraints of quality of service which can be contradictory.

Thus, existing solutions for DSL-based autonomic software deployment does not allow deployment designers to express properties related to multiscale concerns, or only in a limited way concerning some scales from the network or device viewpoints. Additionally, dynamics and openness are not or little considered. Even if MADME deals with the dynamics of the deployment domain, the adaptation of the deployment plan is centralized. Finally, the solutions do not define a complete workflow from the design to the fulfilment of the deployment plan while taking into account the current operational context.

III. Example of the deployment of a multiscale software system

In this section, we present an example of the deployment of a multiscale software system, in order to illustrate our aim. Let’s consider a software system made of different components, each of them having specific individual runtime constraints (memory, OS, etc.). The deployment designer may want to express not only these constraints, but also some requirements related to the distribution of the components. For instance, the deployment designer may want that (C1…C6 are software components):

- a resource-consuming component C1 runs on a cloud,
- C2 runs on several machines in a given geographical area, *e.g.*, a city,
- C3 runs on the same type of device than C1,
- C4 runs on any smartphone of the domain,
- C5 runs on the same network than C4,
- C5 number of deployed instance is relative to C4 instances, *i.e.*, for three instance of C4 on intance of C5 is deployed,
- C4 runs on any new smartphone entering in the domain at runtime,
- C6 runs on one machine on each city.

Moreover, some components may have constraints to run properly, such as:

- C1 requires that the component C0 is installed and activated locally,
- C2 must run on a Linux OS and an Arduino (single-board micro-controller) must be connected to the hosting device,
- C3 requires 40M of free RAM at activation time (Freespace),
- C5 requires a 100G hard drive (HDSize).

Fig. 4 illustrates such an example.

IV. MuScADeL: a DSL for Multiscale Autonomic Deployment

In this section, we describe by means of an example MuScADeL, our proposition of a DSL dedicated to the autonomic deployment of multiscale distributed systems. Tokens and keywords are presented further

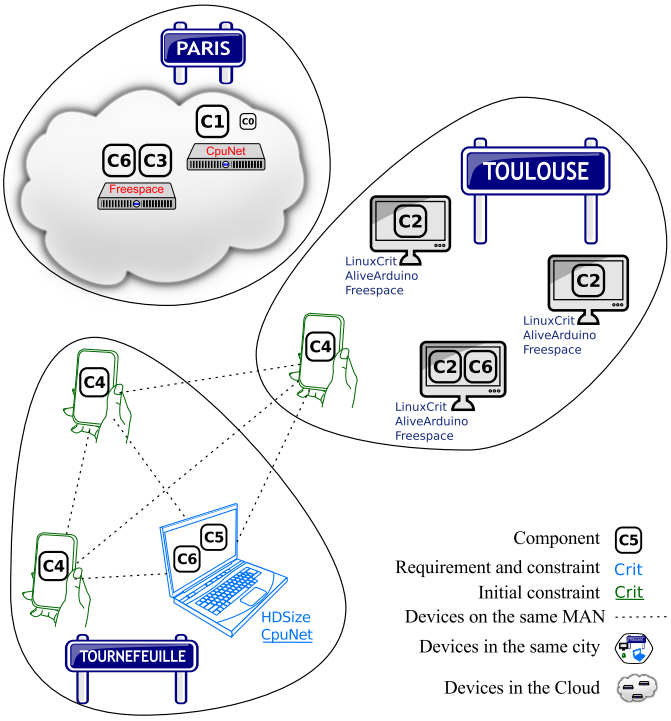

Fig. 4: Example of multiscale deployment.

and the grammar is defined in EBNF syntax (cf. Appendix A). The last version of the grammar is available at http://anr-income.fr/T5/ebnf-muscadel.html.

A. Elements of the language

We present and explain the main elements of MuScADeL language using as example the code for the deployment of the multiscale distributed software system presented in Section III.

1) Component: The keyword **Component** defines a component (cf. Listing 1). The **Version** field is useful for the update activity. The **URL** field specifies the address where the component is reachable for download. The **DeploymentInterface** field specifies the interface of the component, necessary for the interactions with the deployment system: the latter must interact with the component, for configuring and starting it, for managing it at runtime, and for stopping it. The **Dependency** field lists required components: when installing the component, the deployment system checks that whether the required components are installed, or if not, installs them. The **Constraint** field lists hardware and software criteria (defined using the keyword **BCriterion**, see Listing 3) that the component must satisfy. By default, these constraints are permanent —*i.e.*, they must be satisfied both when generating the deployment plan and at runtime — so, the deployment system must check that there is no constraint violation at runtime. For the keyword **InitOnly**, see *6)*.

```
1  Component C0 {
2    Version 1
3    URL "http://test.fr/plopC0.jar"
4  }

6  Component C1 {
7    Version 1
8    URL "http://test.fr/plopC1.jar"
9    Dependency C0
10   DeploymentInterface fr.enac.plop.DIimpl
11 }

13 Component C2 {
14   Version 1
15   URL "http://test.fr/plopC2.jar"
16   DeploymentInterface fr.enac.plop.DIimpl
17   Constraint Freespace LinuxCrit ActiveArduino
18 }

20 Component C3 {
21   Version 1
22   URL "http://test.fr/plopC3.jar"
23   DeploymentInterface fr.enac.plop.DIimpl
24   InitOnly Constraint Freespace
25 }

27 Component C4 {
28   Version 1
29   URL "http://test.fr/plopC4.jar"
30   DeploymentInterface fr.enac.plop.DIimpl
31 }

33 Component C5 {
34   Version 5
35   URL "http://test.fr/plopC5.jar"
36   Constraint HDSize
37   InitOnly Constraint CpuNet
38 }

40 Component C6 {
41   Version 1
42   URL "http://test.fr/plopC6.jar"
43 }
```

Listing 1: **Component** definition in MuScADeL.

2) Probe: The keyword **Probe** defines a probe (cf. Listing 2). A probe has two fields. The first one, the **ProbeInterface**, specifies the interface of the probe. This interface is needed for interactions with the deployment system for information retrieval. The second one, the **URL**, specifies the address where the probe is reachable for download.

```
1  Probe Arduino {
2    ProbeInterface fr.irit.arduino.DIimpl
3    URL "http://irit.fr/INCOME/arduinoProbe.jar"
4  }
```

Listing 2: **Probe** definition in MuScADeL.

3) BCriterion: The keyword **BCriterion** defines a criterion (cf. Listing 3). A criterion is a conjunction of conditions concerning probed values, like in CpuNet (Listing 3, line 14). There are two kinds of conditions concerning either the existence or liveliness of a probe, or a specific value given by a probe. In the first case, the condition is composed by the probe name and the keywords **Exists** or **Active**, which are defined for any probe interface. For example, in Listing 3, at line 2 and 3,

the used probe is `Arduino`, and conditions use default methods **`Exists`** and **`Active`**. In the second case, the condition is composed by the probe name, the method to call, a comparator, and a value. In this case, the method is probe-specific, and defined in the probe interface. For example, in Listing 3 at line 11, the used probe is `RAM`, the information method used is `freeSpace`, and its value is compared to the number `40`, for 40Mb. A criterion can be used to define both a component constraint (cf. Listing 3, line 37) or a deployment requirement (cf. Listing 5, line 4).

```
1  BCriterion ActiveArduino {
2    Arduino Exists;
3    Arduino Active;
4  }

6  BCriterion LinuxCrit {
7    OS.name = "Linux";            //OS probe
8  }

10 BCriterion Freespace {
11   RAM.freeSpace >= 40;          //RAM probe
12 }

14 BCriterion CpuNet {
15   CPU.load < 80;                //CPU probe
16   Network.bandWith > 1024;      //Network probe
17 }

19 BCriterion HDSize {
20   HD.size > 100;                //HD probe
21 }
```

Listing 3: **`BCriterion`** definition in MuScADeL.

4) Multiscale Probe: The keyword **`MultiScaleProbe`** defines a multiscale probe, useful for deployment requirements (cf. Listing 4). Like **`Probe`**, it has only two fields: **`MultiScaleProbeInterface`** and **`URL`**. A specific keyword is necessary because basic and multiscale probes are considered in a different way when generating the deployment plan. At runtime, a multiscale probe allows to identify the scale or the scale instance of their host device in a given viewpoint/dimension/measure.

```
1 MultiScaleProbe Geography {
2   MultiScaleProbeInterface
3     eu.telecom-sudparis.GeographyProbeImpl
4   URL "http://it-sudparis.eu/INCOME/GeoProbe.jar"
5 }
```

Listing 4: **`MultiScaleProbe`** definition in MuScADeL.

5) Deployment: The keyword **`Deployment`** defines the deployment requirements (cf. Listing 5). The keyword **`AllHosts`** allows to specify and delimit the deployment domain: line 2 expresses that the deployment covers all hosts which satisfy the basic criterion `LinuxCrit`. The operator `@` allows to specify deployment requirement specific to a component. These requirements can take several forms:

- The device hosting the component `C1` must satisfy `CpuNet` and be on the scale `Device.StorageCapacity.Giga` (line 4);
- the component `C2` must be deployed on `2` to `4` devices, in the city `Toulouse` (line 5);
- the component `C3` (line 6) must be deployed on one device (implicit) which has the same value in the dimension `Device.Type` as the device hosting `C1`;
- the component `C4` must be deployed on all devices of the scale `Device.Type.Smartphone`, *i.e.*, on all smartphones of the domain (line 7);
- the component `C5` must be deployed on a device which is situated in the same medium area network (MAN) as the device hosting `C4`, the ratio expression `1/3` specifying that there should be one instance of the component `C5` deployed for three instances of the component `C4` (line 8);
- one instance of the component `C6` must be deployed on each scale instance of the scale `Geography.location.City` (line 9).

```
1  Deployment {
2    AllHosts LinuxCrit;

4    C1 @ CpuNet, Device.StorageCapacity.Giga;
5    C2 @ 2..4, Geography.Location.City("Toulouse");
6    C3 @ SameValue Device.Type(C1);
7    C4 @ All, Device.Type.SmartPhone;
8    C5 @ 1/3 C4, SameValue Network.Type.MAN(C4);
9    C6 @ Each Geography.Location.City;
10 }
```

Listing 5: **`Deployment`** definition in MuScADeL.

The keyword **`DifferentValue`** allows to specify the contrary of **`SameValue`**. Using these keywords, it is possible to define a requirement related to a scale or a scale instance.

6) Dynamics and openness: Some constructions of the DSL are particularly well-adapted for the expression of properties related to dynamics and openness. By default, the properties should be satisfied during the entire application runtime, and so must be checked dynamically. The keyword **`InitOnly`** is used to specify that a constraint should be satisfied initially by the generated deployment plan, but maybe not satisfied at runtime. When specifying deployment requirements, the keyword **`All`** allows to specify that a component should be deployed on a subdomain which satisfies (even dynamically) a requirement. In the example, the component `C4` should be deployed on every smartphone of the domain, including those which enter in the domain at runtime; so, the deployment plan evolves dynamically depending on entering and leaving devices.

As the code can be split in several files, the keyword **`Include`** permits to include other files. One of these file must contain the expression of the deployment.

B. Implementation

Using *Xtext* and *Xtend* frameworks (for lexical and syntactic analysis, translation, and generation of Java source code) [19], we have realized an Eclipse plugin for

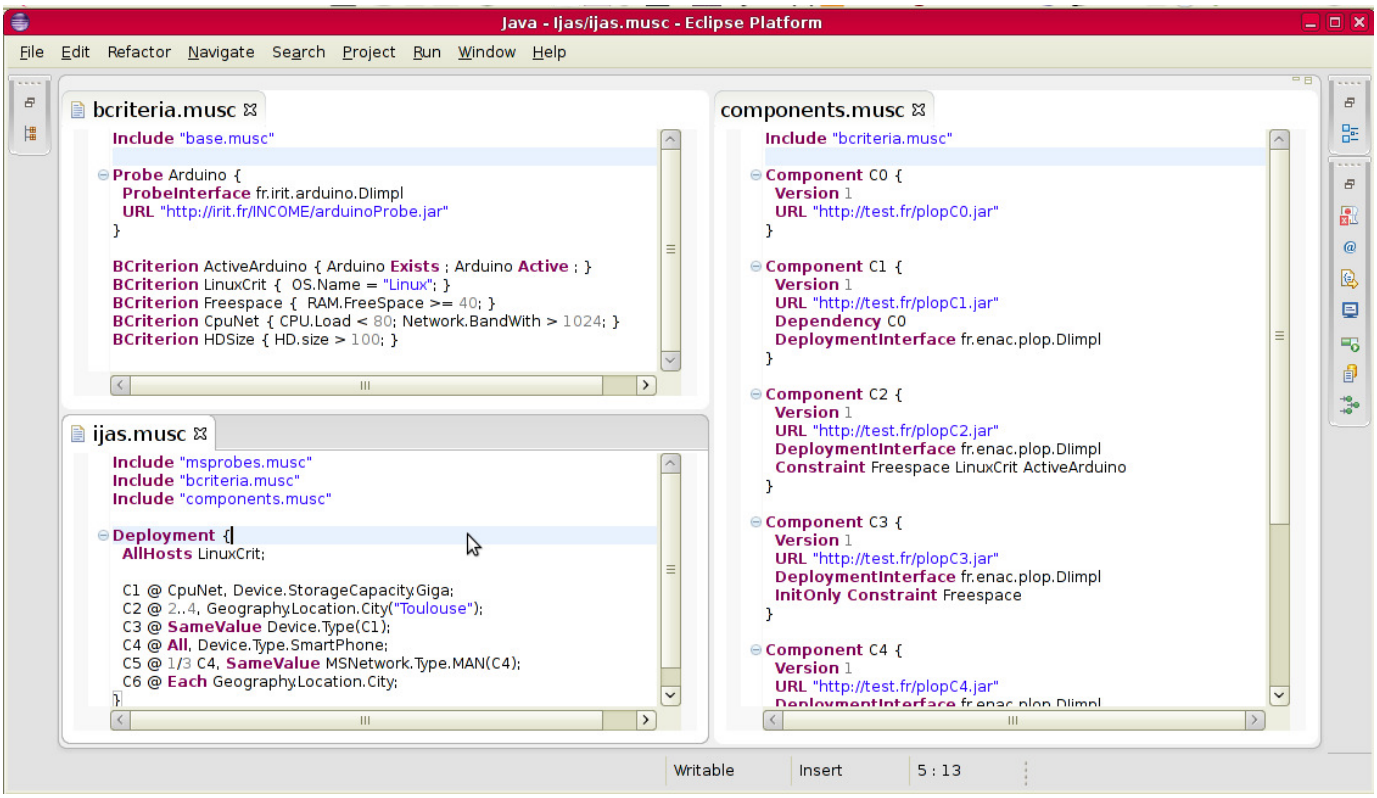

Fig. 5: MuScADeL editor.

the edition of MuScADeL. Using Java and Eclipse makes MuScADeL editor multi-platform compliant and easy-to-use for the deployment designer. Moreover, it runs alongside MuScA (***Mu**ltiscale distributed systems **Sc**ale **A**wareness framework*), a multiscale characterization process, allowing the deployment designer to be able within the same engineering tool to define new multiscale viewpoints, dimensions or scales, before using them in MuScADeL. This binding allows MuScADeL editor to propose an autocompletion of multiscale dimensions and scales and a check of their use. A screenshot of the MuScADeL plugin on Eclipse is shown in Figure 5.

V. From deployment design to a deployment plan

The deployment designer describes the deployment properties using MuScADeL, and then the deployment operator runs the generation of the deployment plan. This generation is a complex process, done in several steps. It is described with a SPEM-like process diagram in Figure 6.

Firstly, the MuScADeL code is compiled, giving in result a file containing the set of components properties, and a file containing a list of probes. The probes are pieces of software that can gather information about a device, such as those described with the *Bcriterion* idiom in the DSL (available memory, OS...). They must be deployed on each host before the deployment, this step being one of the pre-activation activity described in Figure 3.

To achieve the deployment of these probes, we use a light program called *bootstrap*, already installed on each host of the deployment domain. It contains a set of common probes (called later basic probes), and can dynamically acquire and run any other probe specified during this step. Then, the probes are invoked on each device and the results are sent back to the deployment management system.

The set of information gathered on each device is called the *domain state*, representing at that time a view of the status of each device in the deployment domain.

Finally, the constraint solver takes this domain state and the set of properties, and compute a deployment plan as output. Note that we are looking for the first available solution, not to optimize in any way the deployment plan. In case of the constraint solver can not find a solution, the deployment operator is notified and the deployment designer has to change his code.

This process, from MuScADeL edition to a generated deployment plan, is illustrated in a demonstration movie available at http://anr-income.fr/T5/MuScADeL_IDE_Deployment_Plan_Generation.mkv.

The following sections will describe some of the software elements needed to complete the deployment plan, such as the bootstrap architecture and solving step.

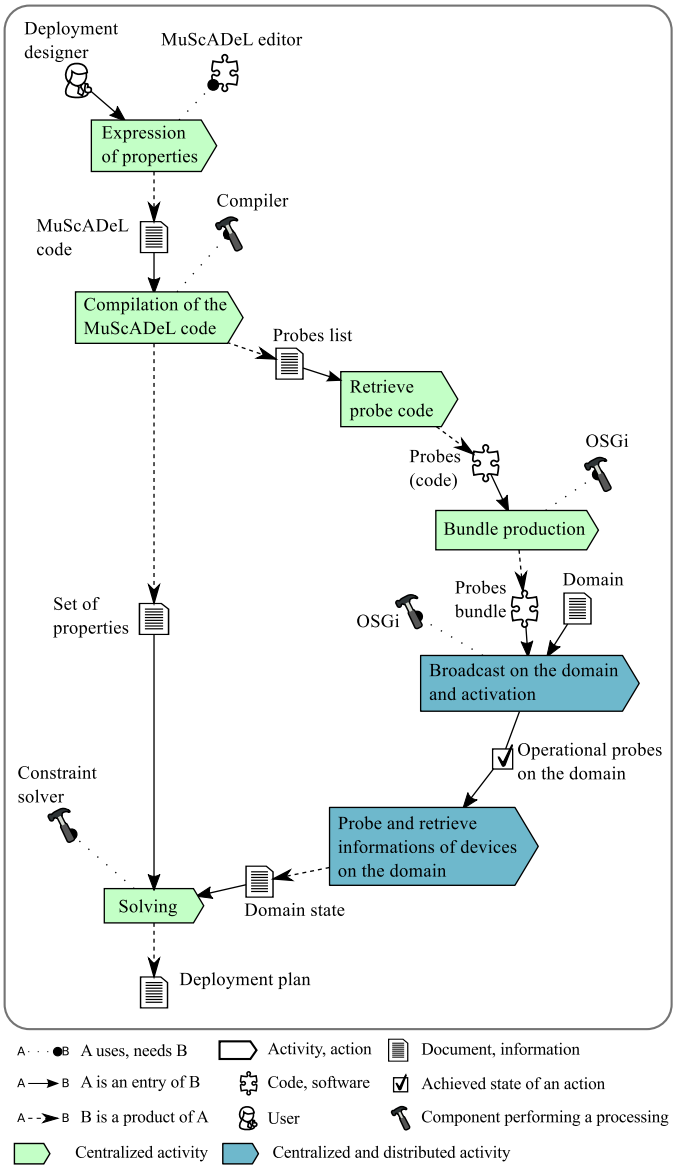

Fig. 6: Generation of the deployment plan.

VI. Bootstrap architecture

In this section, we present the bootstrap, its architecture, and the deployment management system interface (GUI).

A. Bootstrap

The bootstrap of the Deployment Management System (DMS) is a small program available on all devices belonging to the deployment domain. The bootstrap is an OSGi framework containing four bundles: Main, RabbitMQ Client, Basic Probes, and WebService DMS (cf. Figure 7). The Main bundle is the entry point of the bootstrap, and contains the core features of the bootstrap. The RabbitMQ Client bundle insures the link between a device and the deployment monitor. The deployment monitor is a centralized component, in charge of the initial deployment, *e.g.*, probe sending, solving steps, etc., and allows the deployment operator to interact with the DMS. The RabbitMQ Client is useful for the detection of devices appearance and disappearance. The Basic Probes bundle is a set of basic probes, the most common ones. The WebService DMS bundle allows the bootstrap to communicate directly with the deployment monitor. For specific needs, bundles are added to the bootstrap. This extension turns the bootstrap into the device local entity of the DMS.

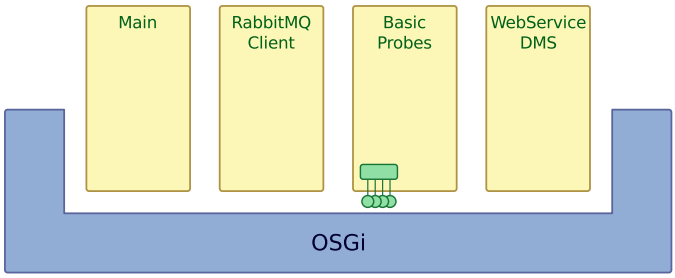

Fig. 7: Bootstrap architecture.

B. Main

The main bundle is the entry point of the bootstrap. It uses the RabbitMQ Client to offer a presence indicator system, and it permits an asynchronous communication system with devices over the network (through firewalls, router, etc.). In this way, it is possible to remotely install, activate, or stop a bundle, or ask for devices state by activating the basic probes service.

C. Basic probes

The Basic Probes bundle contains seven probes, the most useful ones:

- CPU: processor frequency, etc.;
- RAM: free RAM available, full capacity of the RAM, etc.;
- hard disk: full capacity, space available, etc.;
- OS: the operating system, the version of the operating system, etc.;
- network: IP, type of the connection (*e.g.*, Ethernet, WiFi, 3G), etc.;
- and the locale.

Once the deployment monitor sends a request for information, this bundle sends back a packet, on JSON format, containing the result of the probing. An example of returned packet is shown in Listing 6.

D. Android and J2SE

There is two versions of the bootstrap, on in standard Java (J2SE), and another for Android. They are distributed on their native format: *jar* for the J2SE version, and *apk* for the Android version. They contain a start of the OSGi implementation, and the control of the execution. It is the only part of the deployment framework which is not heterogeneous.

```
{
 "basicStatus": {
  "IP_GATEWAY": "81.252.230.88",
  "JAVA_VENDOR": "Oracle Corporation",
  "INTERFACE_LIST": [
   {"name": "wlan0",
    "isUp": true,
    "isLoop": false,
    "ipList": [
     "fe80:0:0:0:e206:e6ff:fecd:4b52%3",
     "10.0.1.146"
    ]},
   {"name": "eth0",
    "isUp": true,
    "isLoop": false,
    "ipList": [
     "fe80:0:0:0:3e97:eff:fe5e:c0db%2",
     "10.0.0.119"
    ]}
  ],
  "CPU_AVAILABLE_PROC": 4,
  "MEM_TOTAL": 115,
  "OS_NAME": "Linux",
  "HD_TOTALSPACE": 179949,
  "MEM_FREE": 92,
  "LOCALE_LANG": "francais",
  "JAVA_NAME": "Java Platform API Specification",
  "HD_FREESPACE": 91082,
  "OS_ARCH": "amd64",
  "OS_VERSION": "3.8.0-34-generic",
  "CPU_NAME": "Intel(R) Core(TM)
               i5-3210M CPU@2.50GHz",
  "JAVA_VERSION": "1.7",
  "JVM_NAME": "Java HotSpot(TM)
               64-Bit Server VM",
  "CPU_SPEED": 1200,
  "LOCALE_COUNTRY": "France",
  "CPU_LOAD": 0.08
 }
}
```


Listing 6: Example of packet returned by Basic Probes bundle on JSON format.

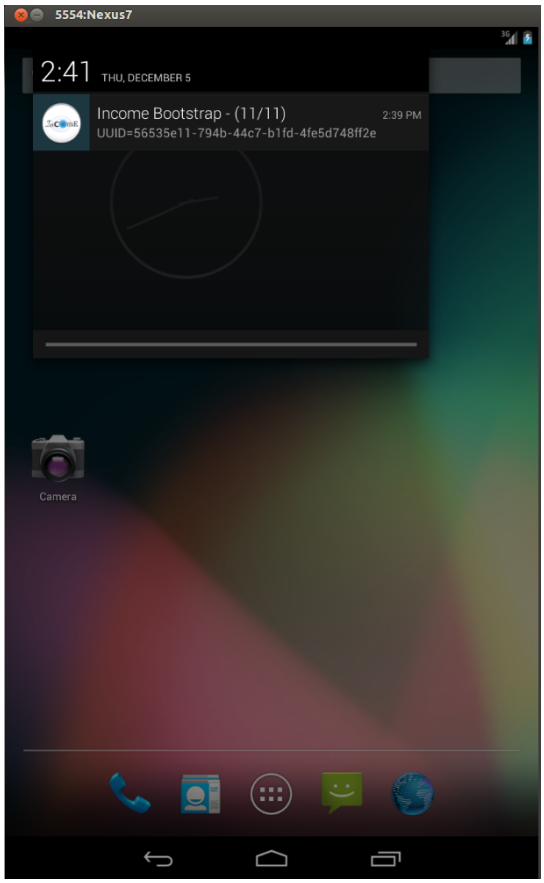

Fig. 8: Android bootstrap - Notification bar.

In both cases, the bootstrap uses the system to indicate to the user that the framework is on. An icon is shown on the bar task, or the notification bar for Android, which allows the user to check the state of the deployment system, stop it, etc. Figure 8 and Figure 9 show pictures of the Android bootstrap. In Figure 8, the notification bar shows the bootstrap icon, and in Figure 9, the bootstrap application interface shows information about installed bundles (full bundles and their status, 32 is for active bundle).

E. DMSMaster

The DMSMaster is a Java software which shows the list of reachable devices in real time, and can ask them to send information about their state (resulting from basic probes). It is a preliminary draft of the final deployment software GUI.

Figure 10 shows two devices running a bootstrap: a 3G connected smartphone and a WiFi connected tablet. The DMSMaster (red circled) lists these two devices and retrieved information.

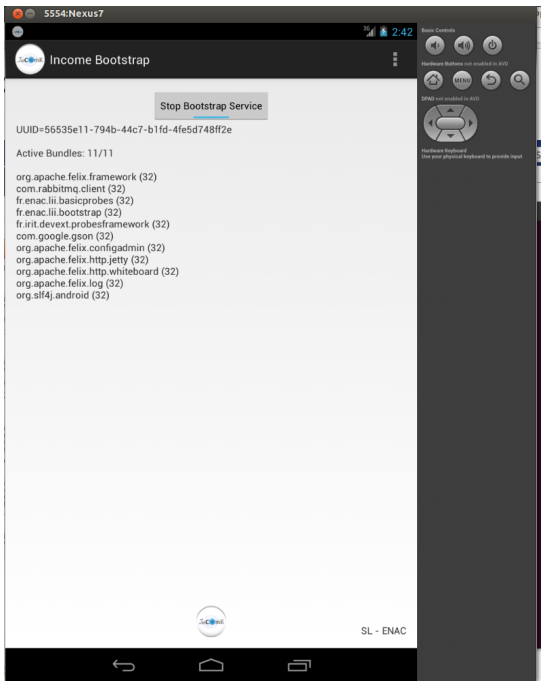

Fig. 9: Android bootstrap interface.

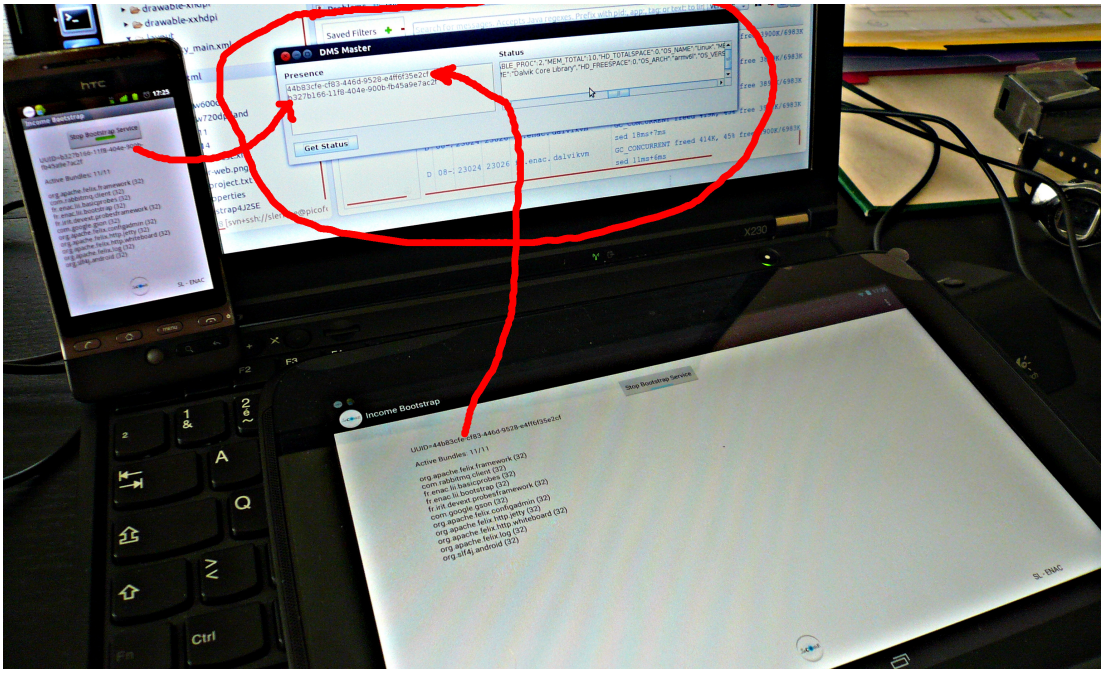

Fig. 10: Picture of the DMSMaster.

VII. Constraints formalization

In this section, we present the formalization of deployment properties.

A. Data and data structure

1) Input data: Analysis of the MuScADeL code allows to identify some properties that the system have to hold. This analysis produces a $Component \times Property$ matrix named $Comp$, defined by

- $Comp(i,j) = 1$ if the deployment of the component $Component_i$ is constrained by the property $Property_j$,
- $Comp(i,j) = 0$ otherwise.

Note that components that are specified to be required for the deployment of another component are taken into account and integrated to the matrix $Comp$. Only simple properties are taken into account for the calculation of $Comp$. Simple properties are basic criteria and multiscale criteria that concern only one component.

On the other hand, independently from the MuScADeL code, analysis of the deployment domain produces a $Device \times Property$ matrix named Dom, defined by

- $Dom(i,j) = 1$ if the device $Device_i$ has property $Property_j$,
- $Dom(i,j) = 0$ otherwise.

Basic and multiscale probes are used to produce the matrix Dom. By and large, measures of devices that are taken by the probes are part of the domain state. They are supplied as an array associating devices and measures.

2) Output data: The deployment plan produced by the solver is a $Component \times Device$ obligation matrix named $Oblig$, defining the placement of each component. It is defined by

- $Oblig(i,j) = 1$ if component $Component_i$ must be deployed on device $Device_j$,
- $Oblig(i,j) = 0$ otherwise.

B. Deployment properties

1) Constraints and requirements: A $Component \times Device$ type possibility matrix named $SatVar$, is build. Each coefficient of $SatVar$ is a variable which can take its value in $\{0,1\}$.

Constraints are added on some coefficients from matrices $Comp$ and Dom. Those constraints are the assignment of the coefficient to the value 0. This assignment correspond to the impossibility for the device to host the component. It is expressed using the following constraint[1] (nb_dev and nb_comp respectively correspond to the number of devices and to the number of components involved in the deployment):

$$\forall i \in \{1,..,nb\_comp\}, \forall j \in \{1,..,nb\_dev\}$$
$$Comp(i) \cdot Dom(j) = \vec{0} \implies SatVar(i,j) = 0 \quad (1)$$

In this formula, $Comp(i)$ and $Dom(j)$ respectively represent rows i and j of matrices $Comp$ and Dom, the operator $\cdot$ constructs a vector composed by the two by two element product of the two given lines, and $\vec{0}$ represents the null vector.

[1]By convention, indexes of row and column matrices and of arrays begin at 1.

2) Number of component instances:

a) Cardinality: For each component (a row of matrix *SatVar*), the number of component instances is the sum of the row's elements. A constraint is then built for each component depending the number of instances.

Thus, if component C_k must be deployed on n_k devices, it would be translated using constraint:

$$\sum_{j=1}^{nb\_dev} SatVar(k,j) = n_k \quad (2)$$

If component C_k must be deployed on n_k to m_k devices, it would be translated using constraint:

$$n_k \leq \sum_{j=1}^{nb\_dev} SatVar(k,j) \leq m_k \quad (3)$$

b) All: The All cardinality specifies that a component must be deployed on all devices that can host it. The number of these devices should be maximized. The expression `Ck @ All` would be translated using following formula:

$$\max_{\substack{j\in\{1,..,nb\_dev\} \\ SatVar(k,j)}} \left(\sum_{i=1}^{nb\_dev} SatVar(k,i) \right) \quad (4)$$

c) Ratio: A ratio between instances of different components can be translated using the same principle than simple cardinality, associating several rows of *SatVar*. The expression `Ck @ n/m Cl` would be translated using constraint:

$$\sum_{i=1}^{nb\_comp} SatVar(k,i) = n \times \left\lfloor \frac{\sum_{i=1}^{nb\_comp} SatVar(l,i)}{m} \right\rfloor \quad (5)$$

where $\lfloor \cdot \rfloor$ refers to floor function.

d) Dependency: When the deployment designer defines a component, he can specify if a component requires another one, by means of keyword **`Dependency`**. In this case, these two components must be on the same device. Suppose component C_k requires component C_l, this would be translated using following formula:

$$\forall i \in \{1,..,nb\_comp\} \quad SatVar(k,i) = 1 \implies SatVar(l,i) = 1 \quad (6)$$

3) Multiscale properties:

a) Dependant components: Multiscale properties expressed by means of the keywords **`SameValue`** and **`DifferentValue`** are defined on several component. Those properties express required conditions for the deployment of components, and are directed by the values provided by the referred multiscale probe. For example, the expression `Ck @ SameValue Some.MS.Scale(Cl)` expresses that instances of components C_k and C_l must be on the same scale instance of `Some.MS.Scale`. Let *MSProbe* be an array which associates for each device the measure of the multiscale probe. It expresses that C_k and C_l are respectively deployed on D_i and D_j only if D_i and D_j have the same value on *MSProbe*, which is:

$$\forall i,j \in \{1,..,nb\_dev\} \quad (SatVar(k,i) \wedge SatVar(l,j)) \implies (MSProbe(i) = MSProbe(j)) \quad (7)$$

b) Placement by scale instance: Finally, a component instance presence on a given scale (expressed by means of the keyword **`Each`**) is defined by a constraint similar to the previous. The set of available devices is limited and identified from measured values by the referred multiscale probe. For example, the expression `Ck @ Each Some.MS.Scale` expresses that one instance of the component C_k must be deployed on each scale instance of `Some.MS.Scale`. In order to do that, two array are required: *MSprobe*, which associates to each device the measure of the required multiscale probe, and *MSProbeId*, which lists unique identifiers of each scale instance. Previous expression would be translated using the constraint (*nb_inst* refers to the number of scale instance):

$$\forall i \in \{1,..,nb\_inst\} \quad \left(\sum_{\substack{j\in\{1,..,nb\_dev\} \\ MSProbeId(i)=MSProbe(j)}} SatVar(k,j) \right) = 1 \quad (8)$$

VIII. MuScADeL Solving

In this section, we present the application of our formalization through the MuScADeL code given in Section IV. Thereafter we present our choice of the constraint solver. Finally we present our library of constraint formalization and its use.

A. Matrices definition

Table Ia gives an exemple of the matrix *Comp* built from the MuScADeL code presented in Section IV. Table Ib gives an example of a matrix *Dom* extracted from the domain state. In these matrices, properties P_1, P_2, P_3, P_4, P_5 et P_6 respectively refer to criteria `CpuNet`, `HDSize`, `Freespace`, `LinuxCrit`, `ActiveArduino`, `Device.Type.Smartphone`, `Device.StorageCapacity.Giga`, and `Geography.Location.City("Toulouse")`.

Note that basic and multiscale probes are used to build *Dom* matrix. Generally, probed measures from devices are part of the domain state. They are provided as an

TABLE I: Data on components and devices.

(a) Component matrix *Comp*.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
C_0	1	0	0	1	0	0	1	0
C_1	1	0	0	1	0	0	1	0
C_2	0	0	1	1	1	0	0	1
C_3	0	0	1	1	0	0	0	0
C_4	0	0	0	1	0	1	0	0
C_5	1	1	0	1	0	0	0	0
C_6	0	0	0	1	0	0	0	0

(b) Device matrix *Dom*.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
D_1	1	1	1	1	0	1	0	0
D_2	1	1	1	1	0	1	0	0
D_3	1	1	1	1	0	1	0	0
D_4	1	1	1	1	0	1	0	1
D_5	1	1	1	1	0	0	1	0
D_6	0	1	0	1	0	0	1	0
D_7	0	1	1	1	0	0	1	0
D_8	1	1	1	1	1	0	0	1
D_9	1	1	1	1	1	0	0	1
D_{10}	1	1	1	1	1	0	0	1
D_{11}	1	1	1	0	1	0	0	1
D_{12}	1	1	1	1	0	1	0	1

TABLE II: Probed data from multiscale probes.

(a) Probed data from `Device`.

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}
`Device.Type` Scale	Smartphone	Smartphone	Smartphone	Smartphone	Server	Server	Server	PC	PC	PC	PC	Smartphone

(b) Probed data from `MSNetwork`.

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}
`MAN`	betelgeuse	betelgeuse	betelgeuse	persee	orion	orion	orion	persee	persee	persee	persee	persee

(c) Probed data from `Geography`.

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}
`City`	Tournefeuille	Tournefeuille	Tournefeuille	Toulouse	Paris	Paris	Paris	Toulouse	Toulouse	Toulouse	Toulouse	Toulouse

TABLE III: Obligation matrix *Oblig*.

	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}	D_{12}
C_0	0	0	0	0	1	0	0	0	0	0	0	0
C_1	0	0	0	0	1	0	0	0	0	0	0	0
C_2	0	0	0	0	0	0	0	1	1	1	0	0
C_3	0	0	0	0	0	0	1	0	0	0	0	0
C_4	1	1	1	0	0	0	0	0	0	0	0	0
C_5	0	0	1	0	0	0	0	0	0	0	0	0
C_6	0	0	1	0	0	0	1	0	0	0	0	1

array associating the device to the measure. For this example, the solving needs information on device type, network identification, and geolocation. The probing is performed respectively by probes `Device`, `MSNetwork`, and `Geography`: probe `Device` identifies the type of the device using the `Type` dimension, probe `MSNetwork` determinate in which medium area network the device belongs to using the scale `Type.MAN`, and probe `Geography` locate in which city is the device using the scale `Location.City`. They produce respectively Tables IIa, IIb, and IIc.

Table III presents a possible obligation matrix, *i.e.*, a deployment plan for the MuScADeL code given in Section IV, probed data from multiscale probes `Device` (cf. Table IIa), `MSNetwork` (cf. Table IIb), and `Geography` (cf. Table IIc).

B. Constraint solver

For the generation of the deployment plan, a constraint solver is used. We had to make a choice on which one to use. Table IV depicts a comparison of constraint solvers. We choose for this comparison: Cream [20], Copris [21], JaCoP [22], or-tools [23], jOpt [24], and Choco [25]. All of them are Java compatible, either written in Java, either can be interfaced with Java. There are different kinds of problem that are handled by constraint programming. Constraint solvers are specialized on several kinds of problems, because their solving is treated differently. In Table IV, acronyms CSP, COP, CP, and JS are respectively constraint satisfaction problem, constraint optimization problem, constraint problem, and job scheduling. We are not specialized on constraint solving problem, and look for a constraint solver easy to use. We compare constraint solvers according the

kinds of problem handled, if they are maintained or deprecated, and if the documentation is up-to-date, and helpful. We do not compare their resolution time because the generation of deployment plan is not crucial on time.

TABLE IV: Constraint solvers comparison.

	Problem	Maintenance	Documentation
Cream	CSP	deprecated (2008)	light
Copris	COP,CSP,CP	maintained	almost nonexistent
JaCoP	CSP	maintained	existent
or-tools	CSP	maintained	almost nonexistent
jOpt	CSP,JS	maintained	missing
Choco	CSP	maintained	complete

As our problem is a constraint satisfaction problem, the kind of problem is not discriminating. The most pertinent for us is **Choco**. The library is simple to use, with the most complete documentation.

C. MuScADeLSolving library

We present here the library MuScADeLSolving. It is composed by the class `MuscadelSolving` (listing 8), its interface `MuscadelSolvingInter` (listing 7), and an exception class `MuscadelSolvingExc`.

The interface `MuscadelSolvingInter` contains methods for constraint addition: `simpleCardinality`, `intervalCardinality`, `allCardinality`, `ratio`, `sameDevice`, `sameValue`, `differentValue`, and `each`[2]. Method `solving` launches the solving of the problem.

```
1  public interface MuscadelSolvingInter {
2      public void simpleCardinality (int cmp,
3                      int card);
4      public void intervalCardinality(int cmp,
5                      int min, int max);
6      public void allCardinality (int cmp);
7      public void ratio(int cmpP, int cmpS,
8                      int ratioP, int ratioS);
9      public void sameDevice(int cmp, int dependsOn);
10     public void sameValue(int cmp1, int cmp2,
11                     String[] probedData);
12     public void differentValue(int cmp1, int cmp2,
13                     String[] probedData);
14     public void each (int cmp, String[] probedData);
15     public int[][] solving()
16                     throws MuscadelSolvingExc;
17 }
```

Listing 7: Interface `MuscadelSolvingInter`.

The class `MuscadelSolving` contains matrices *Comp* and *Dom*, the Choco model and the possibility matrix *SatVar*. *SatVar* is built at the creation of an object `MuscadelSolving` (the constructor calls method `preprocessing`).

```
1  public class MuscadelSolving
2                  implements MuscadelSolvingInter{
3    private Model model;
4    private IntegerVariable[][] satVar;
5    private int nb_comp, nb_app nb_prop;
6    private int[][] comp, dom;
7    private ArrayList<Integer> toMaximize;

9    public MuscadelSolving (int[][] comp, int[][] dom) {
10     assert(comp.length > 0) :
11         "MuscadelSolving:␣empty␣comp";

13     this.model = new CPModel();
14     this.nb_comp = comp.length;
15     this.nb_app = dom.length;
16     this.nb_prop = comp[0].length;
17     this.satVar = new IntegerVariable[nb_comp][nb_app];
18     this.comp = comp;
19     this.dom = dom;
20     toMaximize = new ArrayList<Integer>();

22     preprocessing();
23   }
```

Listing 8: Class `MuscadelSolving`.

Method `preprocessing` builds the possibility matrix `satVar` and adds to it constraints related to the impossibility for the device to host the component, as described by formula (1).

```
1  private void preprocessing () {
2    int [] buffer = new int[nb_prop];
3    // For each variable the domain is defined
4    int[] values = {0,1};
5    for (int i = 0; i < nb_comp; i++) {
6      for (int j = 0; j < nb_app; j++) {
7        satVar[i][j] =
8          Choco.makeIntVar("var_" + i + "_" + j, values );
9        model.addVariable(satVar[i][j]);
10     }
11   }
12   for (int i = 0; i < nb_comp; i++) {
13     for (int j = 0; j < nb_app; j++) {
14       boolean cont = true;
15       for (int k = 0; k < nb_prop; k++) {
16         if (!cont) break;
17         buffer[k] = comp[i][k] * dom[j][k];
18         cont = cont & (buffer[k] == comp[i][k]);
19       }
20       if (!cont)
21         model.addConstraint(Choco.eq(0, satVar[i][j]));
22     }
23   }
24 }
```

Listing 9: Method `MuscadelSolving.preprocessing`.

Method `simpleCardinality` adds simple cardinality constraint, *e.g.*, as described by the formula (2).

```
1  public void simpleCardinality (int cmp, int card) {
2    model.addConstraint(Choco.eq(card,
3                          Choco.sum(satVar[cmp])));
4  }
```

Listing 10: Method `MuscadelSolving.simpleCardinality`.

Method `intervalCardinality` adds interval cardinality constraints –*e.g.*, in listing 5 at line 5– as described by the formula (3). In addition to constraints, this method adds the row of the given component to the list of `satVar` rows to maximize.

```
1  public void intervalCardinality(int cmp, int min, int max) {
2    model.addConstraint(Choco.leq(min,
3                          Choco.sum(satVar[cmp])));
4    model.addConstraint(Choco.geq(max,
5                          Choco.sum(satVar[cmp])));
6    toMaximize.add(cmp);
7  }
```

Listing 11: Method `MuscadelSolving.intervalCardinality`.

[2]In the Java code, indexes of row and column matrices and of arrays begin at 0.

Method `allCardinality` adds `All` cardinality constraint, as described by the formula (4). A constraint is added to specify that at least one instance of a component must be deployed on the deployment domain, and the row corresponding to the component in matrix `satVar` is added to the list of rows to maximize.

```
1 public void allCardinality (int cmp) {
2   model.addConstraint(Choco.leq(1,
3                       Choco.sum(satVar[cmp])));
4   toMaximize.add(cmp);
5 }
```

Listing 12: Method `MuscadelSolving.allCardinality`.

Method `ratio` adds a ratio constraint between components, as described by the formula (5). It has as parameters the components concerned by the ratio, the numerator and the denominator.

```
1 public void  ratio (int cnum, int cdenom,
2                     int rnum, int rdenom) {
3   Constraint ratio =
4     Choco.eq(Choco.sum(satVar[cnum]),
5       Choco.mult(rnum,
6         Choco.div(Choco.sum(satVar[cdenom]), rdenom)));
7   model.addConstraint(ratio);
8 }
```

Listing 13: Method `MuscadelSolving.ratio`.

Method `sameValue` and `differentValue` add multiscale dependant component constraints, as described by the formula (7). They have as parameters referred components and an array of probed data, *e.g.*, the array in Table IIc.

```
1  public void sameValue (int cmp1, int cmp2,
2                         String[] probedData) {
3    checkValue(cmp1, cmp2, probedData, true);
4  }
5  public void differentValue(int cmp1, int cmp2,
6                             String[] probedData) {
7    checkValue(cmp1, cmp2, probedData, false);
8  }
9  private void checkValue (int cmp1, int cmp2,
10                  String[] probedData, boolean diff) {
11   assert probedData.length == nb_app :
12         "checkValue␣tab␣size␣problem␣!";

14   for (int m1 = 0; m1 < nb_app; m1++) {
15     for (int m2 = 0; m2 < nb_app; m2++) {
16       if (! (diff ^ probedData[m1].
17                 equals(probedData[m2])))
18           continue;
19       model.addConstraint(Choco.geq(1,
20         Choco.plus(satVar[cmp1][m1],
21             satVar[cmp2][m2])));
22     }
23   }
24 }
```

Listing 14: Methods `MuscadelSolving.sameValue` and `MuscadelSolving.differentValue`.

Method `each` adds constraint related to the placement of a component instance by scale instance, as described by the formula (8). It has as parameter the referred component and an array of probed data.

```
1  public void each (int cmp, String[] probedData) {
2    assert probedData.length == nb_app :
3          "Each␣tab␣size␣problem␣!";

5    HashMap<String, ArrayList<Integer>> id =
6      new HashMap<String,ArrayList<Integer>>();
7    // Construction of id/index map
8    for (int i = 0; i < probedData.length; i++) {
9      if (id.containsKey(probedData[i]))
10       id.get(probedData[i]).add(i);
11     else {
12       ArrayList<Integer> ids = new ArrayList<Integer>();
13       ids.add(i);
14       id.put(probedData[i], ids);
15     }
16   }
17   // Constraint addition
18   for (String str : id.keySet()) {
19     IntegerExpressionVariable add=Choco.ZERO;
20     for (Integer index : id.get(str)) {
21       add = Choco.plus(satVar[cmp][index], add);
22     }
23     Constraint check = Choco.eq(1, add);
24     model.addConstraint(check);
25   }
26 }
```

Listing 15: Method `MuscadelSolving.each`.

Method `sameDevice` adds constraint related to component dependency, as described by the formula (6). This dependency is specified at the component definition, as shown in the listing 1 at line 9.

```
1 public void sameDevice (int cmp, int dependsOn) {
2   Constraint[] ors = new Constraint[nb_app];
3   for (int i = 0; i < nb_app; i++) {
4     ors[i] = Choco.eq(2,
5       Choco.plus(satVar[cmp][i], satVar[dependsOn][i]));
6   }
7   model.addConstraint(Choco.or(ors));
8 }
```

Listing 16: Method `MuscadelSolving.sameDevice`.

Method `solving` launches the constraint solver's solving. If there is no row on the matrix `satVar` to maximize, the solving is launched directly. Otherwise, maximization instructions are added to the Choco model, then the solving is launched. Thereafter, the feasibility of the problem is checked: if the problem has no solution an exception `MuscadelSolvingExc` is thrown, otherwise, the first solution is returned.

```
1  public int[][] solving() throws MuscadelSolvingExc {
2    Solver solver = new CPSolver();

4    if (toMaximize.size() == 0) {
5      solver.read(model);
6      solver.solve();
7    } else {
8      int up = nb_app*toMaximize.size();
9      IntegerVariable maxx = Choco.makeIntVar("max", 1, up);
10     IntegerExpressionVariable add = Choco.ZERO;
11     for (Integer all : toMaximize) {
12       add = Choco.plus(add, Choco.sum(satVar[all]));
13     }
14     model.addConstraint(Choco.eq(maxx, add));
15     solver.read(model);
16     solver.maximize(solver.getVar(maxx),true);
17   }

19   try{
20     if (solver.isFeasible()) {
21       int [][] result = new int[nb_comp][nb_app];
22       for (int i = 0; i < nb_comp; i++) {
23         for (int j = 0; j < nb_app; j++) {
24           result[i][j] =
25             solver.getVar(satVar[i][j]).getVal();
26         }
27       }
28       return result;
29     } else {
30       throw (new MuscadelSolvingExc("No␣solution")); }
```

```
31    } catch (NullPointerException e) {
32      throw (new MuscadelSolvingExc("No solution"));}
33  }
```

Listing 17: Method `MuscadelSolving.solving`.

D. MuScADeLSolving library use

The class `IJAS` (listing 18) presents the constraint addition phase of the exemple presented in Section IV, listing 5. It contains the main method that builds matrices *Comp* and *Dom* (for the example they are given and not generated by the MuScADeL code analysis), calls MuScADeLSolving methods to add specific constraints, launches the solving, and prints the resulting matrix *Oblig*. The console output is shown in listing 19. This program represents the calculation of the deployment plan of the MuScADeL code presented in Section IV, listing 5.

```
1   public class IJAS {
2     static void printOblig(int[][]oblig) { ... }
3     public static void main(String[] args) {
4       int [][] comp = {
5           { 1, 0, 0, 1, 0, 0, 1, 0 },
6           { 1, 0, 0, 1, 0, 0, 1, 0 },
7           { 0, 0, 1, 1, 1, 0, 0, 1 },
8           { 0, 0, 1, 1, 0, 0, 0, 0 },
9           { 0, 0, 0, 1, 0, 1, 0, 0 },
10          { 1, 1, 0, 1, 0, 0, 0, 0 },
11          { 0, 0, 0, 1, 0, 0, 0, 0 },
12      };
13      int[][] dom = {
14          { 1, 1, 1, 1, 0, 1, 0, 0 },
15          { 1, 1, 1, 1, 0, 1, 0, 0 },
16          { 1, 1, 1, 1, 0, 1, 0, 0 },
17          { 1, 1, 1, 1, 0, 1, 0, 1 },
18          { 1, 1, 1, 1, 0, 0, 1, 0 },
19          { 0, 1, 0, 1, 0, 0, 1, 0 },
20          { 0, 1, 1, 1, 0, 0, 1, 0 },
21          { 1, 1, 1, 1, 1, 0, 0, 1 },
22          { 1, 1, 1, 1, 1, 0, 0, 1 },
23          { 1, 1, 1, 1, 1, 0, 0, 1 },
24          { 1, 1, 1, 0, 1, 0, 0, 1 },
25          { 1, 1, 1, 1, 0, 1, 0, 1 }
26      };

28      System.out.println(
29          "\tGeneration of the deployment plan");
30      MuscadelSolving solv =
31          new MuscadelSolving(comp, dom);

33      //C1 @ CpuNet, Device.StorageCapacity.Giga;
34      solv.simpleCardinality(1, 1);

36      // CO requirements are the same than C1;
37      solv.sameDevice(1,0);
38      solv.simpleCardinality(0, 1);

40      // C2 @ 2..4, Geography.Location.City("Toulouse");
41      solv.intervalCardinality(2, 2, 4);

43      // C3 @ SameValue Device.Type(C1);
44      solv.simpleCardinality(3, 1);
45      String[] deviceType =
46        { "Smartphone", "Smartphone", "Smartphone",
47          "Smartphone", "Server", "Server", "Server",
48          "PC", "PC", "PC", "PC", "Smartphone" };
49      solv.sameValue(3, 1, deviceType);

51      // C4 @ All, Device.Type.Smartphone;
52      solv.allCardinality(4);

54      // C5 @ 1/3 C4, SameValue MSNetwork.Type.MAN(C4);
55      solv.ratio(5, 4, 1, 3);
56      String[] man =
57        { "betelgeuse", "betelgeuse", "betelgeuse",
58          "persee", "orion", "orion", "orion", "persee",
59          "persee", "persee", "persee", "persee" };
60      solv.sameValue(5, 4, man);

62      // C6 @ Each Geography.Location.City;
63      String[] cities =
64        { "Tournefeuille", "Tournefeuille", "Tournefeuille",
65          "Toulouse", "Paris", "Paris", "Paris", "Toulouse",
66          "Toulouse", "Toulouse", "Toulouse", "Toulouse" };
67      solv.each(6, cities);

69      try {
70        int[][] oblig = solv.solving();
71        printOblig(oblig);
72      } catch (MuscadelSolvingExc e) {
73        System.err.println("Problem during the solving.");
74      }
75    }
76  }
```

Listing 18: Main class `IJAS`.

```
1   Generation of the deployment plan
2   Oblig :
3   C0 : 0 0 0 0 1 0 0 0 0 0 0 0
4   C1 : 0 0 0 0 1 0 0 0 0 0 0 0
5   C2 : 0 0 0 0 0 0 0 1 1 1 0 0
6   C3 : 0 0 0 0 0 0 1 0 0 0 0 0
7   C4 : 1 1 1 0 0 0 0 0 0 0 0 0
8   C5 : 0 0 1 0 0 0 0 0 0 0 0 0
9   C6 : 0 0 1 0 0 0 1 0 0 0 0 1
```

Listing 19: Console output.

IX. Conclusion and future work

In this paper, we firstly present MuScADeL, a DSL for multiscale and autonomic deployment, and explain the various elements of the language by means of an example. Then, we present how the deployment plan is computed, using a compiler, and a constraint solver. MuScADeL allows to express the deployment properties of a multiscale software system and its components. These properties drive the computation of the deployment plan, and are used by the autonomic deployment system do detect (and possibly repair) any property violation at the application runtime.

Another part of our work concerns the realization of this autonomic deployment system. We are designing it as a middleware, on the same basis than first experiments described in our previous work [9]. This middleware will enable deployment in multiscale environments. It provides the probes needed to gather information about the hosts.

We believe that a DSL is the best way for a deployment designer to describe deployment requirements and constraints. A DSL has much more expressiveness than any Markup Language (such as XML), and is more efficient since the deployment designer expresses (and read) directly concepts of its field of expertise. Moreover, modern tools for making DSL allows their designers to integrate several level of validation (not only syntactic but also semantic).

Presently, MuScADeL targets the installation and activation activities. Other activities and features, as property infringement at application runtime, are hard coded

in the deployment management system. We plan to move some of them at the DSL level, to increase expressiveness and flexibility when designing deployment. For example, we can add in the grammar the keyword **`on-deinstall`** or **`on-update`** to define actions to perform when deinstalling or updating a component.

Focusing on multiscale systems, we do need a sound and extensible vocabulary to describe the dimensions and their scales. In the INCOME project, another ongoing work aims at defining an ontology for multiscale distributed systems. We continuously integrate these concepts in MuScADeL.

Besides, we are currently working on a toolchain for our DSL. Using Xtext and Xtend frameworks [19], we have realized an Eclipse plugin for the edition of the DSL that makes it multi-platform compliant and easy-to-use for a deployment designer. We have also realized a compiler and a solving algorithm to generate the deployment plan. Using this IDE and the compiler, the deployment designer expresses deployment properties, and launches the generation of the deployment plan. The DSL, the Eclipse plugin, the compiler, and the solving algorithm are part of the deliverables of the INCOME project.

Acknowledgments

This work is part of the French National Research Agency (ANR) project INCOME [5] (ANR-11-INFR-009, 2012-2015). The authors thank all the members of the project that contributed directly or indirectly to this paper.

Appendix

This appendix presents the EBNF syntax of MuScADeL.

⟨*root*⟩ ::= ⟨*muscadel-element*⟩+

⟨*muscadel-element*⟩ ::= ⟨*include*⟩
| ⟨*probe*⟩
| ⟨*bcriterion*⟩
| ⟨*component*⟩
| ⟨*msprobe*⟩
| ⟨*deployment*⟩

⟨*include*⟩ ::= 'Include' '"' ⟨*file-id*⟩ '"'

⟨*probe*⟩ ::= 'Probe' ⟨*probe-id*⟩ '{'
'ProbeInterface' ⟨*interface*⟩
('URL' ⟨*string*⟩)?
'}'

⟨*probe-id*⟩ ::= ⟨*id*⟩

⟨*interface*⟩ ::= ⟨*interface-id*⟩ ('.' ⟨*interface-id*⟩)*

⟨*interface-id*⟩ ::= ⟨*id*⟩

⟨*bcriterion*⟩ ::= 'BCriterion' ⟨*bcriterion-id*⟩ '{'
(⟨*condition*⟩ ';')+
'}'

⟨*bcriterion-id*⟩ ::= ⟨*id*⟩

⟨*condition*⟩ ::= ⟨*probe-id*⟩ '.' ⟨*method-id*⟩ ⟨*comp*⟩
⟨*probe-value*⟩
| ⟨*probe-id*⟩ 'Exists'
| ⟨*probe-id*⟩ 'Active'

⟨*method-id*⟩ ::= ⟨*id*⟩

⟨*probe-value*⟩ = ⟨*int*⟩
| ⟨*string*⟩

⟨*comp*⟩ ::= '<' | '>' | '<=' | '>=' | '='

⟨*component*⟩ ::= 'Component' ⟨*component-id*⟩ '{'
'Version' ⟨*int*⟩
'URL' ⟨*string*⟩
('DeploymentInterface' ⟨*interface*⟩)?
('Dependency' (⟨*component-id*⟩)+)?
('InitOnly'? 'Constraint' ⟨*bcriterion-id*⟩)*
'}'

⟨*component-id*⟩ ::= ⟨*id*⟩

⟨*msprobe*⟩ ::= 'MultiScaleProbe' ⟨*msprobe-id*⟩ '{'
'MultiScaleProbeInterface' ⟨*interface*⟩
'URL' ⟨*string*⟩
'}'

⟨*ms-probe-id*⟩ ::= ⟨*id*⟩

⟨*deployment*⟩ ::= 'Deployment' '{'
('AllHosts' (⟨*bcriterion-id*⟩)+ ';')?
(⟨*deployment-requirement*⟩ ';')+
'}'

⟨*deployment-requirement*⟩ ::= ⟨*component-id*⟩ '@'
⟨*requirement-rhs*⟩ (',' ⟨*requirement-rhs*⟩)+
';'

⟨*requirement-rhs*⟩ ::= 'Each' ⟨*mscriterion-scale*⟩
| 'SameValue' ⟨*mscriterion-dependency*⟩
| 'DifferentValue' ⟨*mscriterion-dependency*⟩
| ⟨*mscriterion*⟩
| ⟨*bcriterion-id*⟩
| ⟨*ratio*⟩
| ⟨*location*⟩
| ⟨*cardinality*⟩

⟨*mscriterion-dependency*⟩ ::= ⟨*msprobe-id*⟩ '.' ⟨*dim-id*⟩ '('
⟨*component-id*⟩ ')'
| ⟨*msprobe-id*⟩ '.' ⟨*dim-id*⟩ '.' ⟨*scale-id*⟩ '('
⟨*component-id*⟩ ')'

⟨*mscriterion-scale*⟩ ::= ⟨*msprobe-id*⟩ '.' ⟨*dim-id*⟩ '.' ⟨*scale-id*⟩

⟨*mscriterion*⟩ ::= ⟨*mscriterion-scale*⟩
| ⟨*msprobe-id*⟩ '.' ⟨*dim-id*⟩ '.' ⟨*scale-id*⟩ '(' ⟨*string*⟩ ')'

⟨*dim-id*⟩ ::= ⟨*id*⟩

⟨*sc-id*⟩ ::= ⟨*id*⟩

⟨*ratio*⟩ ::= ⟨*int*⟩ '/' ⟨*int*⟩ ⟨*component-id*⟩

⟨*location*⟩ ::= ⟨*int*⟩ '.' ⟨*int*⟩ '.' ⟨*int*⟩ '.' ⟨*int*⟩

⟨*cardinality*⟩ ::= ⟨*int*⟩
| ⟨*interval*⟩
| 'All'

⟨*interval*⟩ ::= ⟨*int*⟩ '..' ⟨*int*⟩

References

[1] R. Boujbel, S. Leriche, and J.-P. Arcangeli, "A DSL for multi-scale and autonomic software deployment," in International Conference on Software Engineering Advances (ICSEA 2013), L. Lavazza, R. Oberhauser, A. Martin, J. Hassine, M. Gebhart, and M. Jäntti, Eds., 2013, pp. 291–296.

[2] G. Blair and P. Grace, "Emergent middleware: Tackling the interoperability problem," IEEE Internet Computing, vol. 16, no. 1, pp. 78–82, jan.-feb. 2012.

[3] M. Kessis, C. Roncancio, and A. Lefebvre, "DASIMA: A flexible management middleware in multi-scale contexts," in 6th Int. Conf. on Information Technology: New Generations (ITNG '09), april 2009, pp. 1390–1396.

[4] M. van Steen, G. Pierre, and S. Voulgaris, "Challenges in very large distributed systems," Journal of Internet Services and Applications, vol. 3, no. 1, pp. 59–66, 2012.

[5] J.-P. Arcangeli, A. Bouzeghoub, V. Camps, C. M.-F. Canut, S. Chabridon, D. Conan, T. Desprats, R. Laborde, E. Lavinal, S. Leriche, H. Maurel, A. Péninou, C. Taconet, and P. Zaraté, "INCOME - Multi-scale context management for the Internet of Things," in Ambient Intelligence, 3rd Int. Joint Conf. AmI 2012, ser. Lecture Notes in Computer Science, F. Paternò, B. E. R. d. Ruyter, P. Markopoulos, C. Santoro, E. v. Loenen, and K. Luyten, Eds., vol. 7683. Springer, 2012, pp. 338–347.

[6] S. Rottenberg, S. Leriche, C. Lecocq, and C. Taconet, "Vers une définition d'un système réparti multi-échelle," in Journées francophones Mobilité et Ubiquité (UBIMOB). Cépaduès Editions, 2012, In French.

[7] S. Rottenberg, S. Leriche, C. Taconet, C. Lecocq, and T. Desprats, "From Smartdust to Cloud: The emergence of multiscale distributed systems," 2013, Unpublished Paper.

[8] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf, "A characterization framework for software deployment technologies," Defense Technical Information Center (DTIC) Document, Tech. Rep., april 1998.

[9] M. E. A. Matougui and S. Leriche, "A middleware architecture for autonomic software deployment," in The 7th Int. Conf. on Systems and Networks Communications (ICSNC'12). Lisbon, Portugal: XPS, 2012, pp. 13–20, 12619 12619. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00755352

[10] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," Computer, vol. 36, no. 1, pp. 41–50, 2003.

[11] A. Van Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," ACM Sigplan Notices, vol. 35, no. 6, pp. 26–36, 2000.

[12] M. Strembeck and U. Zdun, "An approach for the systematic development of domain-specific languages," Software: Practice and Experience, vol. 39, no. 15, pp. 1253–1292, 2009.

[13] J.-P. Tolvanen and S. Kelly, "Integrating models with domain-specific modeling languages," in Proceedings of the 10th Workshop on Domain-Specific Modeling, ser. DSM '10. New York, NY, USA: ACM, 2010, pp. 10–1. [Online]. Available: 10.1145/2060329.2060354

[14] A. Flissi, J. Dubus, N. Dolet, and P. Merle, "Deploying on the grid with deployware," in CCGRID. IEEE Computer Society, 2008, pp. 177–184.

[15] A. Dearle, G. N. C. Kirby, and A. J. McCarthy, "A framework for constraint-based deployment and autonomic management of distributed applications," in International Conference on Autonomic Computing (ICAC'04). IEEE Computer Society, 2004, pp. 300–301.

[16] A. Dearle, G. N. C. Kirby, and A. McCarthy, "A middleware framework for constraint-based deployment and autonomic management of distributed applications," CoRR, vol. abs/1006.4733, 2010.

[17] K. Sledziewski, B. Bordbar, and R. Anane, "A DSL-based approach to software development and deployment on cloud," in 24th IEEE Int. Conf. on Advanced Information Networking and Applications (AINA 2010). IEEE Computer Society, 2010, pp. 414–421.

[18] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 73–100, 2012.

[19] M. Eysholdt and H. Behrens, "Xtext: implement your language faster than the quick and dirty way," in SPLASH/OOPSLA Companion, ser. Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA, W. R. Cook, S. Clarke, and M. C. Rinard, Eds. ACM, 2010, pp. 307–309.

[20] N. Tamura, "Cream: Class library for constraint programming in Java," last access: June 2014. [Online]. Available: http://bach.istc.kobe-u.ac.jp/cream

[21] ——, "Copris: Constraint Programming in Scala," last access: June 2014. [Online]. Available: http://bach.istc.kobe-u.ac.jp/copris/

[22] K. Kuchcinski and R. Szymanek, "JaCoP - Java constraint programming solver," last access: June 2014. [Online]. Available: http://jacop.osolpro.com/

[23] "or-tools, operations research tools developed at Google," last access: June 2014. [Online]. Available: https://code.google.com/p/or-tools/

[24] "jOpt, Java OPL implementation," last access: June 2014. [Online]. Available: http://jopt.sourceforge.net/opl.php

[25] C.H.O.C.O. Team, "CHOCO: an open source Java constraint programming library," Ecole des Mines de Nantes, Tech. Rep. 10-02-INFO, 2010, last access: June 2014. [Online]. Available: http://www.emn.fr/z-info/choco-solver/