A DSL for multi-scale and autonomic software deployment

Raja Boujbel, Sébastien Leriche, Jean-Paul Arcangeli

A DSL for Multi-Scale and Autonomic Software Deployment

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

Sébastien LERICHE
Université de Toulouse
ENAC
7 av. Edouard Belin
31055 Toulouse, France
Email: 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—**In this paper, we present an ongoing work which aims at defining and experimenting a domain-specific language (DSL) dedicated to multi-scale and autonomic software deployment. Autonomic software deployment in open environments is an open issue. There, the topology of target hosts is not always known due either to unforeseen hardware failures or limitations (network links, hosts...) or to device arrival and disappearance. In a previous work, we proposed to describe deployment constraints using a DSL and then to satisfy them using a middleware for autonomic deployment, rather than classically building and executing a deployment plan. As deployment of multi-scale distributed systems demands the expression of specific constraints related to dimensions and scales, it is necessary to think over and define a new domain-specific language. In this paper, we propose a new DSL designed to support the expression of constraints and properties related to multi-scale and autonomic software deployment.**

Keywords*—*Deployment, Multi-Scale, DSL, Component-Based Software System

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 to build systems which are distributed over several scales, called "multi-scale" systems.

The INCOME project [1] aims at designing software solutions for multi-scale context management, not only in ambient networks but also in the Internet of Things and clouds, 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 multi-scale applications which must be deployed, *i.e.* made and kept available for use, in a situation of mobility and variability of the quality of the resources. In this project, our work focuses on software deployment and our goal is to develop a framework for supporting the deployment of multi-scale applications such as context managers. Deployment strategies should take into account the multi-scale aspects like geography, network, device, and user, as well as non functional properties such as efficiency and privacy. In multi-scale systems, decentralization, autonomy and adaptiveness are essential features.

In this paper, we present an ongoing work which aims at defining and experimenting a Domain-Specific Language (DSL) dedicated to multi-scale and autonomic software deployment.

The paper is structured as follows. Section II introduces the two main aspects of our working context: multi-scale distributed systems and software deployment. Section III provides an example of deployment of a multi-scale software system, analyses the requirements, and proposes to use a DSL to support autonomic deployment. Section IV discusses related work on DSL for software deployment. Our DSL is presented in Section V using the example presented in Section III. Section VI concludes and discusses some perspectives.

II. Context Overview

This section introduces the novel concept of multi-scale system and provides an overview of software deployment.

A. Multi-scale distributed systems

The term "multi-scale system" is present in several recent research papers [2], [8], [16]: 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.

In [10], authors argue that the multi-scale nature of a distributed system should be analyzed independently in several specific dimensions such as geography, network, device, data, user... Thus, a distributed system can be described as multi-scale when, for at least one dimension, the elements of its projection onto this dimension are associated with different scales. Figure 1, extracted from [11], shows an example of scales in the "Device processing power" dimension.

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

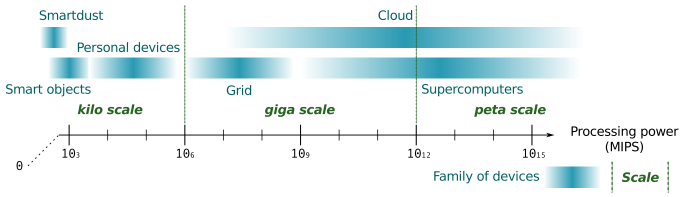

Fig. 1. Scales in the "Device processing power" dimension

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 [3]. Figure 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 runtime.

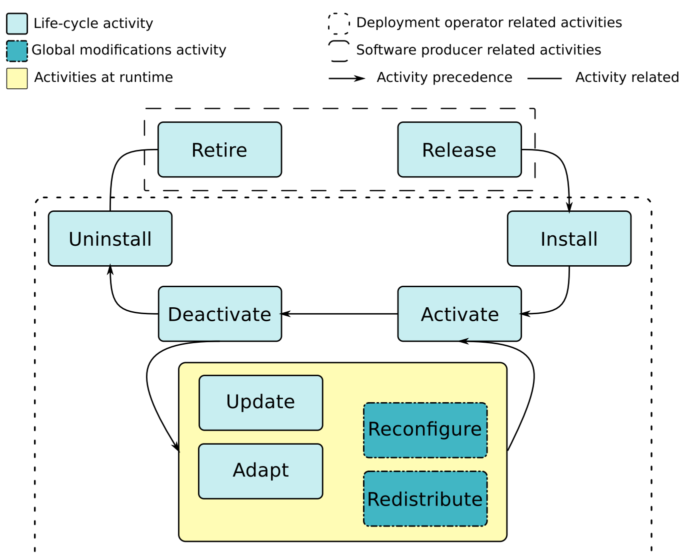

Fig. 2. Software deployment life cycle

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 properties (commands, 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 constraints. 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. The deployment domain is a distributed set of machines which host the software system and provides resources to it. The ultimate purpose of deployment design is to produce a deployment plan which complies with the expressed properties and constraints. 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.

Figure 3 shows the timeline of deployment.

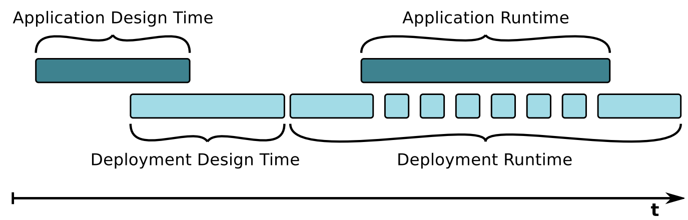

Fig. 3. Software deployment timeline

III. Deployment of multi-scale software systems

In this work, we focus on the design phase of the deployment process, and precisely on the ways for a deployment designer to express deployment properties and constraints.

Here is an example, in order to illustrate our aim. Let's consider a software system made of different components, each of them having specific individual runtime requirements (memory, OS…). The deployment designer may want to express not only these requirements, but also some other ones related to the distribution of the components. For instance, the deployment designer may want that (C1…C5 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 device than C1,
- C4 runs on any smartphone of the domain,
- C5 runs on the same network than C4,
- C4 runs on any new smartphone entering in the domain at runtime.

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 microcontroller) must be connected to the hosting device,
- C3 requires 40M of free RAM at activation time (Constr1),
- C5 requires a 100G hard drive (Constr2).

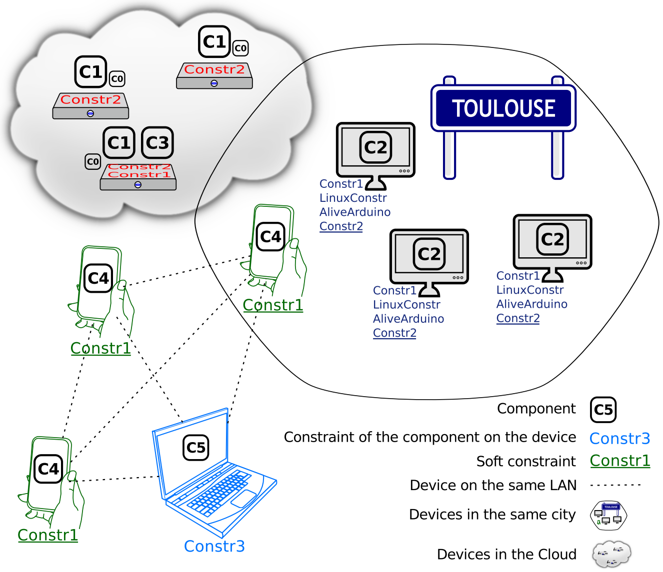

Fig. 4. Example of multi-scale deployment

Figure 4 illustrates such an example.

This section analyses the problem of software deployment of multi-scale systems from the design point of view, and then motivates the use of a Domain-Specific Language (DSL) which supports the expression of multi-scale deployment properties and constraints.

A. Analysis

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.

In addition, 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 favour of autonomy: the autonomic computing approach [7], where the system self-manages some properties (self-configuration, self-healing), may support solutions which satisfy the requirements of distributed multi-scale software systems deployment. This idea lead us to "autonomic software deployment" [9].

B. Our approach

Instead of directly expressing a statically defined deployment plan, we propose to express deployment constraints and properties from which the deployment plan can be computed. In this paper, we focus on the expression of the constraints and properties, not on the construction of the plan. For this last point, our idea is use a constraint solver, supplying it with an up-to-date description of a domain (available hosts and their properties).

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 multi-scale (dimensions and scales). Relations between probes and properties can be made explicit at the same level as the deployment properties and constraints in order to allow the specification of the system of probes at the deployment design time.

C. Towards a DSL for autonomic software deployment of multi-scale systems

In this ongoing work, our aim is to provide a solution for the expression of the deployment design, concerning in particular the dimensions and other significant properties of multi-scale software systems.

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 and constraints. So, we propose a domain-specific language (DSL) dedicated to the description of deployment constraints and properties. 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 [15], [13], [14].

IV. Related work on DSL for software deployment

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.

Dearle *et al.* [5], [4] 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 multi-scale properties and constraints. Openness is neither taken into account, the set of hosts is statically defined in a file by the deployment manager. 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.* [12] present an approach that incorporate 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, this model is translated into specific code and automatically deployed onto the Cloud. This approach is specific to deployment on the cloud. It highlights the need to facilitate the work of the deployment designer, and that using DSL is a solution for that.

V. Proposition of a DSL

In this section, we describe by means of an example our proposition of a DSL dedicated to the autonomic deployment of multi-scale distributed systems. Tokens and keywords are presented further and the grammar is defined in EBNF syntax[1].

A. Example

We give below a full example of code for the deployment of the multi-scale distributed software system presented in Section III. Then, we use this code to present and explain the main elements of the language.

```
Include "base.jasd"
//base.jasd defines some probes
//like OS, RAM, CPU, Network, and HD

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

Component C1 {
    Version 1
    URL "http://test.fr/plopC1.jar"
    Require C0
    DeploymentInterface fr.enac.plop.DIimpl
}

Probe Arduino {
    ProbeInterface fr.irit.arduino.DIimpl
    URL "http://irit.fr/INCOME/arduinoProbe.jar"
}

Constraint AliveArduino {
    Arduino Exist, Alive
}

Constraint LinuxCstr {
    OS.Name = "Linux"            //OS probe
}

Constraint Constr1 {
    RAM.FreeSpace >= 40          //RAM probe
}

Constraint Constr2 {
    CPU.Load < 80                //CPU probe
    Network.BandWith > 1024      //Network probe
}

Constraint Constr3 {
    HD.size > 100                //HD probe
}

Component C2 {
    Version 1
    URL "http://test.fr/plopC2.jar"
    DeploymentInterface fr.enac.plop.DIimpl
    Constraint Constr1, LinuxCstr, AliveArduino
    Soft Constraint Constr2
}

Component C3 {
    Version 1
    URL "http://test.fr/plopC3.jar"
    DeploymentInterface fr.enac.plop.DIimpl
    Soft Constraint Constr1
}

Component C4 {
    Version 1
    URL "http://test.fr/plopC4.jar"
    DeploymentInterface fr.enac.plop.DIimpl
    Soft Constraint Constr1, Constr2
}

Component C5 {
    Version 2
    URL "http://irit.fr/plopC5.jar"
    Constraint Constr3
}

MultiScaleProbe Geography {
  MultiScaleProbeInterface
    eu.telecom-sudparis.GeographyProbeImpl
  URL "http://it-sudparis.eu/INCOME/GeoProbe.jar"
}

//other MultiScale probes are described
//the same way
//{...}

Deployment {
    AllHosts LinuxCstr

    C1 @ Constr2, Device.Cloud
    C2 @ 2..4 Geography.City("Toulouse")
    C3 @ SameValue Device(C1)
    C4 @ All Device.SmartPhone
    C5 @ SameValue Network.MAN(C4)
}
```

B. Elements of the language

1) Component: The keyword **`Component`** defines a component. 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

[1] The grammar is available at http://www.irit.fr/~Raja.Boujbel/ebnf-jasd.html

interact with the component, for configuring and starting it, for managing it at runtime, and for stopping it. The **Require** field lists required components: at installation time of the component, if the required component is not installed, the deployment system must install it on the device. The **Constraint** field lists hardware and software constraints of the component. By default, these constraints are hard, *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). For the keyword **Soft**, see *6)*.

2) Probe: The keyword **Probe** defines a probe. A probe has two mandatory 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.

3) Constraint: The keyword **Constraint** defines a constraint on a component. It has one kind of field, a probe value test. There can be several tests in a **Constraint**, like in `Constr2` (line 35). A probe value test is composed by two or three parts. If the constraint is related to the existence or the liveliness of a hardware or a software component, the probe value test is composed by the probe name and keywords **Exists** or **Alive**. These keywords are defined for any probe interface. For example, at line 23, the used probe is `Arduino`, and the constraint uses default methods **Exists** and **Alive**. If the constraint is about a value, the probe value is composed by the probe name, a method call, a comparator, and a value. There, the method is probe specific, and defined in the probe interface For example, at line 28, the used probe is `OS`, the information method used is `Name`, and its value is compared to the string `"Linux"`.

4) Multi-scale Probe: The keyword **MultiScaleProbe** defines a multi-scale probe, useful for the deployment. Like **Probe**, it has only two fields. The first one, **MultiScaleProbeInterface**, specifies the interface of the probe. The second one, **URL** specifies the address where the implementation of the probe is reachable for download. In our current solution, scales are defined in the implementation of probes, and the probes allows to identify the scale of a given device.

5) Deployment: The keyword **Deployment** defines the deployment properties and constraints. The keyword **AllHosts** allows to specify and delimit the deployment domain: line 83 expresses that the deployment covers all hosts which satisfy the constraint `LinuxCstr`. The operator `@` allows to specify deployment constraints specific to a component. These constraints can take several forms: the device hosting the component `C1` must satisfy `Constr2` and be on the scale `Cloud` on the dimension `Device` (line 85); the component `C2` must be deployed on `2` to `4` devices, in the city `Toulouse` (line 86); the component `C4` must be deployed on all devices of the dimension `Device.Smartphone`, *i.e.* on all smartphones of the domain (line 88). The keyword **SameValue** expresses that the component must be in the same dimension or scale as a referred one: the component `C3` (line 87) must be deployed on one device (implicit) which has the same value in the dimension `Device` as the device hosting `C1` (in other words, `C3` should be deployed on the same device as `C1`); the component `C5` must be deployed on a device which is situated in the same medium area network (MAN) as the device hosting `C4` (line 89).

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 constraints should be satisfied during the entire application runtime, and so must be checked dynamically. The keyword **Soft** is used to specify that a constraint should be satisfied initially by the generated deployment plan, but maybe not satisfied at runtime. When specifying the **Deployment**, the keyword **All** allows to specify that a component should be deployed on a subdomain which satisfies (even dynamically) a property or a constraint. 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.

The file must have at least one definition of a component and one expression of the deployment. Other fields are optional. As the code can be split in several files, the keyword **Include** permits to include other files (line 1).

VI. CONCLUSION AND FUTURE WORK

In this paper, we present the first version of a DSL for multi-scale and autonomic deployment, and explain the various elements of the language by means of an example. This DSL allows to express the deployment constraints of a multi-scale software system and its components. These constraints drive the computation of the deployment plan, and are used by the autonomic deployment system do detect (and possibly repair) any constraint 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 multi-scale environments. It will provide the probes needed to gather informations about the hosts.

We believe that a DSL is the best way for a deployment designer to describe deployment 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, the DSL targets the installation and activation activities. Other activities and features, as constraint infringement at application runtime, are hard coded in the deployment manager system. In the future, we can 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 multi-scale 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 multi-scale distributed systems. We plan to integrate these concepts in our DSL.

Besides, we are currently working on a toolchain for our DSL. Using Xtext and Xtend frameworks [6], 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. The DSL and the Eclipse plugin are part of the deliverables of the INCOME project.

ACKNOWLEDGMENTS

This work is part of the French National Research Agency (ANR) project INCOME[2] (ANR-11-INFR-009, 2012-2015). The authors thank all the members of the project that contributed directly or indirectly to this paper.

REFERENCES

[1] 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 - Multiscale Context Management for the Internet of Things. In F. Paternò, B. E. R. d. Ruyter, P. Markopoulos, C. Santoro, E. v. Loenen, and K. Luyten, editors, *AmI*, volume 7683 of *Lecture Notes in Computer Science*, pages 338–347. Springer, 2012.

[2] G. Blair and P. Grace. Emergent Middleware: Tackling the Interoperability Problem. *Internet Computing, IEEE*, 16(1):78–82, jan.-feb. 2012.

[3] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek, and A. L. Wolf. A characterization framework for software deployment technologies. Technical report, DTIC Document, 1998.

[4] A. Dearle, G. N. C. Kirby, and A. McCarthy. A Middleware Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications. *CoRR*, abs/1006.4733, 2010.

[5] A. Dearle, G. N. C. Kirby, and A. J. McCarthy. A Framework for Constraint-Based Deployment and Autonomic Management of Distributed Applications. In *ICAC*, 1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA, pages 300–301. IEEE Computer Society, 2004.

[6] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *SPLASH/OOPSLA Companion*, 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, pages 307–309. ACM, 2010.

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

[8] M. Kessis, C. Roncancio, and A. Lefebvre. DASIMA: A Flexible Management Middleware in Multi-Scale Contexts. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pages 1390–1396, april 2009.

[9] M. E. A. Matougui and S. Leriche. A middleware architecture for autonomic software deployment. In *ICSNC '12 : The Seventh International Conference on Systems and Networks Communications*, pages 13–20, Lisbon, Portugal, 2012. XPS. 12619 12619 .

[10] 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.

[11] S. Rotteneberg, S. Leriche, C. Taconet, C. Lecocq, and T. Desprats. From Smartdust to Cloud: The Emergence of Multiscale Distributed Systems. Unpublished Paper, 2013.

[12] K. Sledziewski, B. Bordbar, and R. Anane. A DSL-Based Approach to Software Development and Deployment on Cloud. In *AINA*, 24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Perth, Australia, 20-13 April 2010, pages 414–421. IEEE Computer Society, 2010.

[13] M. Strembeck and U. Zdun. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience*, 39(15):1253–1292, 2009.

[14] J.-P. Tolvanen and S. Kelly. Integrating models with domain-specific modeling languages. In *Proceedings of the 10th Workshop on Domain-Specific Modeling*, DSM '10, pages 10–1, New York, NY, USA, 2010. ACM.

[15] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[16] M. van Steen, G. Pierre, and S. Voulgaris. Challenges in very large distributed systems. *Journal of Internet Services and Applications*, 3(1):59–66, 2012. 10.1007/s13174-011-0043-x.

[2] http://anr-income.fr