Infrastructure as Runtime Models - Towards Model-Driven Resource Management

Filip Krikava, Romain Rouvoy, Lionel Seinturier

Infrastructure as Runtime Models - Towards Model-Driven Resource Management

Filip Křikava,
Czech Technical University, Czech Republic
filip.krikava@fit.cvut.cz

Romain Rouvoy, Lionel Seinturier
Inria / University of Lille, France
first.last@inria.fr

***Abstract*—The importance of continuous delivery and the emergence of tools allowing to treat infrastructure configurations programmatically have revolutionized the way computing resources and software systems are managed. However, these tools keep lacking an explicit model representation of underlying resources making it difficult to introspect, verify or reconfigure the system in response to external events.**

In this paper, we outline a novel approach that treats system infrastructure as explicit runtime models. A key benefit of using such models@run.time representation is that it provides a uniform semantic foundation for resources monitoring and reconfiguration. Adopting models at runtime allows one to integrate different aspects of system management, such as resource monitoring and subsequent verification into an unified view which would otherwise have to be done manually and require to use different tools. It also simplifies the development of various self-adaptation strategies without requiring the engineers and researchers to cope with low-level system complexities.

I. Introduction

The growing scale of computing infrastructures keeps raising new challenges in its provisioning, management, monitoring and recently also self-adaptation [1]. Beyond the scale, the complexity is also increasing, since even a small cluster, deployed privately in an enterprise or formed within some public cloud provider, will include several loosely coupled software components and services. All of which have to be appropriately configured, monitored and adapted during the system execution.

Most of the components and services provide some monitoring and management interfaces. However, despite that conceptually, there is no difference in spawning new virtual machine (*VM*), starting a new service, adding a new user into a system or creating a new database, performing all these actions require use of several tools with different syntax and semantics. This gives rise to accidental complexities not only for system administration but also for developing any self-adaptive behavior as the researchers and engineers are obliged to deal with lot of low-level system details.

The increasing adoption of continuous delivery and the emergence of tools, such as Ansible, Puppet or Chef have revolutionized the way computing resources and software systems are managed.[1] These tools treat the infrastructure configuration programmatically. Using these tools for provisioning and maintenance of server environments resembles the way software engineers build and maintain application source code. Essentially, they can be considered as model transformation tools that take a description—*i.e.* a model—of desired states of a set of computing resources (*e.g.*, virtual machines, services, users, databases) and generate a set of necessary commands that appropriately configure the actual resources. For example, Ansible takes resource state description in a YAML format and generates a set of Python scripts that are executed on managed hosts.

Using such design-time models already yields a significant improvement in the way systems are configured. However, these models have the potential to be also used at runtime. Having an explicit model of computing resources available at runtime brings many additional advantages beyond resource provisioning and reconfiguration. They allow one to integrate different aspects of system management, such as resource monitoring and subsequent system verification into an unified view which would otherwise have to be done manually and require to use different tools. Another key benefit of using such runtime models is that they provide a rich and uniform semantic foundation for computing resources monitoring and reconfiguration [2]. This significantly simplifies building higher-level abstractions for configuration management system self-adaptation.

In this paper, we focus on such explicit runtime models (*model@run.time* [3]) and propose a novel approach for infrastructure management and self-adaptation. The aim is to provide a flexible abstraction that (i) raises the level of abstraction on which the computing resources are monitored and managed, (ii) accurately reflects the state of the resources it represents, and (iii) can be access through a wide range of clients (*e.g.*, from existing modeling tools, command line, domain-specific languages). Furthermore, using models allows one to reuse some well-established techniques, languages and tools from classical *Model-Driven Engineering* (MDE). For example, model verification can be used to check the system consistency, model transformation can export an infrastructure model into a visualization or reporting tool, and model comparison and model merging can be used to discover differences between computing resources and to consolidate them.

II. Motivation

In this section, we motivate the use of *model@run.time* for computing resource management. We do that by presenting some of the drawbacks in the current infrastructure management on a typical system administration task and self-adaptation strategy. We argue that these issues can be alle-

[1] Ansible (http://ansible.com), Puppet (http://puppetlabs.com), Chef (http://chef.io) are the three most popular infrastructure automation systems according to the GitHub ranking as of the time of writing.

viated by shifting from design-time model to *model@run.time* and in the next section we show concrete examples of how these drawbacks are addressed by our vision of resource management.

As a running example, we consider the administration of OpenStack[2], which is a popular open-source cloud computing platform. It is composed of a several interconnected services, such as a compute service (*nova*) for managing VM instances an image service (*glance*) for managing VM images, and a monitoring service (*ceilometer*) for collecting measurements of the utilization of the deployed physical and virtual resources.

A. System Administration

A common administration task is spawning a virtual machines. In Ansible 1.9, this can be realized using its `nova_compute` module with the following code excerpt:

```
- nova_compute:
    state: present
    login_tenant_name: demo
    name: vm1
    flavor_id: 3 # m1.medium
    image_name: ubuntu-14.04-x86_64
    floating-ips: [ 192.168.1.32 ]
```

It specifies a computing resource, a `m1.medium` sized VM called `vm1` within the `demo` cloud tenant, accessible via `192.168.1.32` IP address.

The following is a list of shortcomings that we encounter with this administration task. It is important to note, that the list is by no means complete and it solely reflects our experience. Furthermore, while we choose Ansible as it is the most popular infrastructure automation tool (according to GitHub star ranking), the following issues largely applies to the other tools as well.

— *Lack of Verification.* Upon execution, Ansible translates the model resource state into a Python code, which is consequently executed. Since the transformation happens at design-time without any connection to the target cloud, the possible verification is limited to basic type conformance. However, Ansible cannot ensure that the given IP belongs to the `demo` tenant or whether given image fits the selected VM size.

— *Leaky Idempotence.* Ansible resource definitions are idempotent—*i.e.* no action is performed once an action has put a resource into a desired state. In this example, no new VM will be started if there exists a VM named `vm1`. The problem is that if the administrator changes the VM configuration (*e.g.* flavor, image), none of these changes will be applied. To have them applied, the administrator will have to first tear down the VM manually. Since some changes might affect the VM uptime this might be sometimes the desired behavior, however, the choice should depend on the system administrator. This is particularly true when the infrastructure management is part of a continuous deployment in which committing a change in the resource configuration model shall be immediately reflected in the target infrastructure.

  Furthermore, there are multiple strategies to reflect such changes and their choice depends solely on the system administrator and the state of the system. For example, changing the VM image can be done by either recreating the entire VM or by spawning a new one and live-migrating the original VM state.

— *Lack of Introspection.* One of the root cause of the above issues is the lack of introspection of the target system. This makes one to rely on additional tools to query the state of the resource. The main problem is that these additional tools operate on a different level of abstraction. To give a concrete example, we might want to find out what are the differences between the currently running `vm1` instance and the one we have specified in the above snippet. Currently, this is rather an error-prone task since it results in manually comparing all the VM meta-data.

  Furthermore, the lack of introspection makes the VM definition rather static. For instance, it is not possible to specify that we want to have the largest possible VM within the tenant quotas or to use the newest available Ubuntu image.

— *Lack of Composability.* Imagine a scenario in which we would like to create VM on the first available host that has the lowest temperature. While OpenStack supports scheduling hints to influence how *nova* controller chooses the target host on which a VM will be created, this use case is not yet supported[3]. On the other hand it can be realized using the existing services. The temperature can be collected by Ganglia[4] or many other monitoring tools and *nova* scheduler supports forcing a particular host. However, to realize this in Ansible is not straightforward and requires to create a new module that replicates most of the original `nova_compute` functionality because Ansible modules are not composable.

  Another scenario applicable to our example is cloud meta-scheduling in which a VM could be scheduled in more than one cloud provider. For instance, if the private cloud is fully utilized, we might want to submit a VM instance to one of the public cloud providers. Similarly, this is not a trivial to express due to the lack of composability.

B. Self-Adaptation

A classical[5] example of cloud self-adaptation is auto-scaling. It leverages cloud elasticity allowing a VM cluster to dynamically adjust its size based on its utilization. There are different strategies to achieve the auto-scaling itself (*i.e.* computing the size of VM cluster based on its utilization), for example using utility theory [4] or control theory [5]. However, currently all these solutions rely on a set of custom scripts that collect the necessary metrics (*e.g.*, memory, CPU usage, response time, throughput) and provide the reconfiguration action (*e.g.* enlisting and discharging VMs). These scripts need to query different services and build an ad hoc system model themselves. This requires significant engineering effort and

[2]http://openstack.org

[3]https://www.mail-archive.com/openstack-dev@lists.openstack.org/msg11978.html

[4]http://ganglia.sourceforge.net

[5]Amazon EC2 already provides auto scaling support *cf.* http://aws.amazon.com/autoscaling. In OpenStack, it is planned *cf.* https://blueprints.launchpad.net/heat/+spec/autoscaling-api-resources.

complicates experimenting with self-adaptive behavior as the researchers and engineers need to deal with many low-level system details.

III. Vision of Models@Run.Time for Infrastructure Management

In this section we present our vision of the *model@run.time* for infrastructure management. We start with an overview of the *model@run.time* for our running example and show an illustration how can it be used to introspect and modify the underlying OpenStack resources. Next, we discuss the runtime meta-model and some additional features it supports. For brevity sake, we deliberately omit some technical details.

A. Overview

Figure 1 shows an excerpt of the *model@run.time* for OpenStack management. It contains two top-level models for the *nova* and *glance* services together with models for some of the OpenStack resources. Each reference and attribute additionally contains an annotation specifying its mutability (`r/w`) and whether, changes have to be manually observed (`o`), or change notifications are supported (`n`). Since most of the events in OpenStack get posted in a message queue, the majority of features supports change notifications.

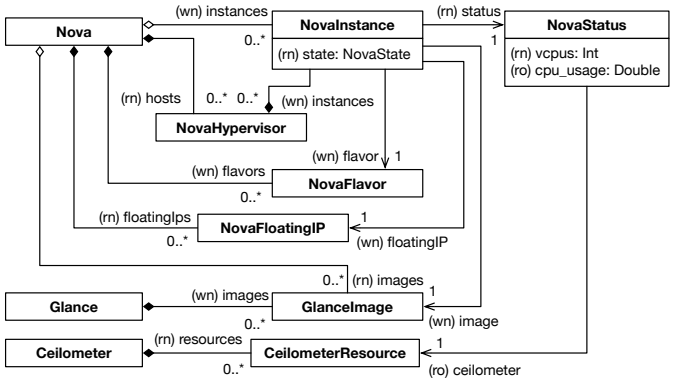

Fig. 1. *Model@run.time* for OpenStack management (excerpt)

B. Introspection and Modification

While it is possible to have a multiple client interfaces (*cf.* Section IV-C), in the following examples we access the *model@run.time* using an internal DSL in Scala[6]. We use the interactive Scala read-eval-print-loop console for the execution environment, but full Scala programs can be used alike. One of the main reason for a DSL is that it operationalizes the *model@run.time* in the sense that it enables manipulating instances of the meta-model. Its syntax is automatically generated from the meta-model definition and aligned with Scala. Furthermore, it allows one to mix declarative model manipulation with imperative code constructs. The reason for choosing Scala is that (i) it supports both object-oriented and functional style of programming, (ii) it uses type inference to combine static type safety with a "*look and feel*" close to dynamically typed languages, and (iii) it has been designed to host DSLs.

[6]http://scala-lang.org

To start manipulating our *model@run.time*, we first need to connect to a *model@run.time* instance.

```
> val nova = connect[Nova]("localhost/nova-demo")
```

This constructs a connection to a given *model@run.time* (*i.e.* `Nova`) deployed at a local endpoint (*i.e.* `/nova-demo`). Once connected, we can introspect the state of the virtual machines by simply navigating the model references and attributes:

```
> nova.instances
| RTList(
|  NovaInstance(name="vm0", state=NovaState.Active,
|    image=GlanceImage(name="ubuntu-14.04-x86_64", ...),
|    flavor=NovaFlavor(id=3, name="m1.medium", ...),
|    status=NovaStatus(vcpu=2, cpu_usage=0.87, ...) ...),
|  NovaInstance(name="vm1", ...), ...)
```

This returns all the deployed virtual machines in a form of causally connected list (`RTList`). Any changes to such a list are immediately propagated to the running system. For example, adding a new element spawns a new VM:

```
> nova.instances += NovaInstance(name = "vm2", ... )
| RTFuture[NovaInstance](...)
```

The return type of this operation (and most of others) is a future object that represents an asynchronous (and potentially long running) operation. It can be used to monitor the operation state and progress if supported by the underlying resource. The `RTList` provides most of the expected Scala collection operations allowing one to easily query the state of the resources in an expressive and concise way (*cf.* Appendix A).

All model instances are also causally connected and changing element features is reflected in the connected resource. For example, changing VM flavor results in a resize action:

```
> instance.flavor := nova.flavors find (_.name=="m1.large")
```

Multiple resize strategies are supported using implicit values[7]. For instance:

```
> implicit val s = NovaResizeStrategy.Recreate
```

overrides the default resize strategy for all the subsequent calls.

C. Meta-Model

The meta-model is similar to the EMF Ecore meta-model [11]. It is statically typed with multiple inheritance and fully reflective. A model class represents a computing resource. It can contain a number of features (*i.e.* attributes and references) and define a number of operations. Similarly to EMF, it also allows single reference containment only and additionally, each instance has a string identifier, which is unique within its container. The structural features further contain an annotation about their mutability and observability.

At the type level, the runtime model differentiates between instances that are causally *connected* to the underlying resources and instances that only *describe* the desired resource state. Let us consider the following example:

```
> val instance = NovaInstance(name = "vm2", ... )
| NovaInstanceDesc(name="vm2")
```

The call to the model class object factory constructs only a description of a desired resource state. This instance is not connected to any resource yet. The connection is established by containing the instance within another already contained one:

[7]Values that are automatically provided to method arguments in the case they are not give explicitly by caller.

```
> nova.instances += instance
// or
> nova.hosts(0).instances += instance
```

Either of these operations will spawn a new VM and create its causally connected instance that can be accessed through the returned future. The difference between the two operations is that the latter will schedule the instance creation with the schedule hint that forces the selected hypervisor (*i.e.* `hosts(0)`). Similarly, removing a model instance from its containment disconnects it and can potentially destroy the underlying computing resource.

The main consequence is that a connected instance cannot escape the boundary of the top level container—*i.e.* the host that runs the model. For example, given two distinct nova services (`nova1` and `nova2`), the following is not allowed:

```
> nova2.instances += nova1.instance(0)
```

On the other hand it is possible to try to recreate a VM:

```
> nova2.instances += nova1.instance(0).copy()
```

The `copy()` operation returns the model instance description.

It is important to note here that changing a containment within the same model is possible. For example:

```
> nova.hosts(1).instances += nova.hosts(0).instances(0)
```

will try to live migrate the first VM from the first hypervisor to the second one.

Having an explicit model of computing resources allows one to use some of the existing MDE tools for its manipulation. For example FUNNYQT [8] or SIGMA [7] can be directly used for some common model manipulation tasks. It is also possible to transform snapshots of *model@run.time* into into a more conventional Ecore models and reuse tools from EMF ecosystem (*e.g.* for comparison, visualization or reporting).

D. Model Composition

The preferred way to create runtime models is through composition. In practice, this means that we first design the model classes of computing resources that reflect the available operations from a single service point of view. These service-level runtime models can be then composed together to create higher-level abstractions that provides a more holistic view of the system. For example, the `NovaStatus` is built using the ceilometer service.

E. Model Monitoring

The model distinguishes between features that are reactive—*i.e.* can post notifications—and features whose changes must be manually observed. The reactive features can be monitored without any additional work. For example:

```
> monitor(nova.instances) {
   case ElementAdded(e) => log("Added new VM: "+e.name)
  }
```

creates a monitor[8] that logs a message every time a new VM is being scheduled.

Features that have to be manually observed with a help of periodic monitor:

```
> monitor(nova.instances(0).status.cpu_usage, 2.sec) { ... }
```

[8]The `monitor` function takes as a second argument a partial function that is executed for all matching events

Instead of a single attribute, an aggregate can be monitored as well (*e.g.* `nova.instances.map(_.status.cpu_usage).mean`).

Monitors are runtime models as well (contained in the top level model element). Consequently, they can be also monitored (*e.g.* changes in the triggering period). Monitors provide the fundamental building block for autonomous self-adaptation as they allow to systematically react to system state changes.

F. Model Consistency Checking

Similarly to the design-time model, *model@run.time* also supports capturing structural constraints as state invariants. An invariant is represented as a boolean query associated to a containing reference. The main difference to a design-time constraints is that the evaluation is triggered automatically at runtime upon every change of the corresponding resource or its *model@run.time* representation.

For example, the following constraint checks whether a given instance can fit the remaining project quota:

```
> constraint("Size within quota", nova.instances) { self =>
  nova.quota.isAvailable(self.flavor)
}
```

Constraints are implemented using monitors. Similarly they are also runtime models and thus they can be also monitored (*e.g.* their violations).

G. Interconnection of Various Runtime Models

So far we have only shown examples within the same *model@run.time* scope. However, an infrastructure usually consists of more than one system and therefore it is important to be able to cross model boundaries. For example, a VM instance might be running a MySQL database. Using our *model@run.time*, it can be accessed using simple traversal:

```
> instance.models(_.name == "mysql-root").as[MySQL].dbs
| RTList(MySQLDB(name="root", tables=RTList(...)))
```

This implies that that the `NovaInstance` inherits from system `Host` which provide the `models` reference discovering deployed models in a given host.

IV. PROOF OF CONCEPT IMPLEMENTATION

This section provides an overview of the implementation concepts of out vision of *model@run.time* for infrastructure management presented above. It consists of three parts: (1) a model definition, (2) an implementation of causal connection, and (3) a synthesis of the runtime platform and client interfaces.

A. Model Definition

The model definition consists of a structural description of the modeled computing resource (*e.g.* Figure 1). Given that our *model@run.time* meta-model is similar to Ecore, the definition is currently done in EMF, reusing the existing Ecore editors. We rely on Ecore annotations to attach additional meta-information to the modeling elements. In the future, we plan to provide a dedicated DSL for better designing experience.

B. Causal Connection

From a model definition, we synthesize a skeleton Scala implementation that has to be completed with a code that *connects* the model features and operations to the underlying computing resources. This phase is currently manual and requires most of the engineering effort. The amount of effort largely depends on the quality of the available interfaces used to monitor and manage the computing resource. The same can be observed from the-state-of-the-art infrastructure automation tools that also relies on a manual orchestration of management interfaces.

For resource-oriented services, there is a potential to automate this phase, at least to some degree, by synthesizing the connectors directly from service descriptors (*e.g.* WADL[9]). Nevertheless, the essential challenge is in being able to accurately represent the state of a resource without having any significant performance impact. This is particularly difficult for resources that do not provide any push style notifications and whose changes have to be therefore manually observed. The current state of the art leaves many opportunities for further research.

C. Runtime Platform and Client Interfaces

Finally, taking the model definition and implementation of causal connectors, we can synthesize the runtime platform and generate client interfaces. The implementation is based on the Actor model [12] using the Akka[10] runtime. The system architecture is a classical client/server. The server deploys one or model instances and associates them with a distinct endpoints to which the clients connect.

Server. The idea is that every model class is handled by an actor. We have therefore a flat hierarchy of actors that correspond to all model definitions that are currently loaded. Each actor handles directly only model attributes and operation invocations while forwarding reference traversals to the corresponding actors of the given reference type. This allows the actors to be stateless and we can use routing patterns that can scale the number of actors according to the number of requests (starting a new actor in Akka is a cheap operation). The only exceptions are monitors, which are implemented as stateful actors (they have references to their trigger allowing them to gracefully shutdown). The memory complexity of the runtime platform is therefore linear to the number of deployed model classes and monitors.

Client. In general, from the client perspective, the model definition is technologically agnostic and therefore there might be potentially many different client interfaces. The only requirement is to be able to make an HTTP connection to the actor runtime. Currently, we focus on a concrete interface in a form of a Scala internal DSL. Essentially, it is a library that allows one to write fragments of code with domain-specific syntax. These fragments are woven within Scala own syntax so that it appears different.The advantage of internal DSL is that it allows us to reuse all the existing language infrastructure and libraries without any additional effort (*e.g.* Scala console and collection operations).

The generated DSL classes are proxies to the remotely running actors. Each connected model instance has additionally two private attributes: an element path and an endpoint. The element path is a unique path that identifies the element instance within a given endpoint (URL to the actor server). It is the combination of the instance unique name together with the name of its containing reference starting at the beginning of the model hierarchy. For example a VM instance `vm` with id `vm0` that is deployed on a hypervisor `host0`:

```
vm = nova.hosts(_.name=="host0").instances(_.name=="vm0")
```

has the following element path: `//nova-demo/hosts/host0/instances/vm0`. Executing `vm.status.cpu_usage` will append `/status/cpu_usage` to the path above and sends a request to the associated endpoint asking for the value of the attribute. The processing of such a request is explained in Figure 2. Basically, each actor consumes a part of the element path and either answers the client in the case the path has been exhausted or further delegates the request to an actor that represents the reference traversal. The reason for this representation is that it allows us to design the system in a purely reactive and asynchronous manner without the need to introduce any blocking operations.

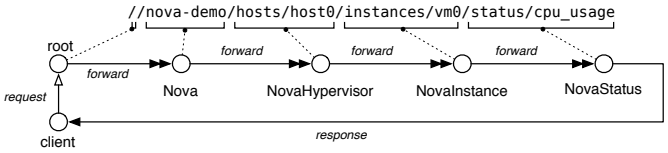

Fig. 2. Message forwarding and delegation.

Another important consequence of this design is that there are no *living* model instances. This implementation gives an *illusion* of a graph of causally connected model instances, however, they are recreated with every request (each actor loads an instance of the model it is responsible for). The reason is that, these instances represent existing resources which are already present in the system and more importantly, which can be externally modified.

V. Discussion and Conclusion

The main motivation for this work is to provide an approach that leverages infrastructure management. While, there has been some preliminary work using *model@run.time* to target parts of specific infrastructures [9], [10], we rely on *model@run.time* to integrate various monitoring and management interfaces together in a logical way, providing a holistic view of the system with an uniform and rich semantic base for its manipulation.

The *model@run.time* provides a foundation on top of which new higher-level abstractions and tools can be built, in particular for infrastructure automation. For example, instead of sequential execution of a resource configuration scripts, the deployment might be treated as an infrastructure adaptation and use some sophisticated control algorithms (*e.g.* discrete control). Similarly, the infrastructure *model@run.time* can be used as the monitoring and reconfiguration layer in some of the self-adaptive software engineering frameworks and

[9]http://www.w3.org/Submission/wadl/

[10]http://akka.io

toolkits (*e.g.*, RAINBOW [4], EUREMA [14] or ACTRESS [6]). Moreover, using a specific client, the infrastructure model could used directly from tools like Matlab. This would greatly simplify the work needed for system identification and controller design.

A. Research Challenges and Limitations

The are many research challenges associated with the use of *model@run.time* [13]. In the context of infrastructure management, we find the following to be the most import ones:

— What is the right abstraction in which a multi-layer, multi-tier system infrastructure can be expressed?
— How to scale such abstraction so it can represent a large number of assets in the targeted domain?
— How to create causal connections that accurately represent the state of the underlying system, yet without any significant performance impact?
— How can this connection be automatically synthesized for existing resource oriented services?
— How to turn legacy services into resource oriented ones?
— How to handle resource inconsistencies and failures?
— Which of the existing MDE tools could be used in connection to such *model@run.time* and how to make the runtime model accessible to them?
— What are the relevant user interfaces that shall be provided to the users to interact with the model?

While this work tries to partially address some of them, much is left for further work. The main limitations is currently in the prototype implementation, in the way the causal connection is implemented (*i.e.* manually), and in the scale of current experiments. We believe that the principal design choice of using reactive platform and the Actor model will allow us to scale the implementation. In Akka 2.0 version, the memory overhead is only about 400 bytes per actor instance (2.7 million actors per GB of heap) with a possible throughput of 50 million messages per sec on a single machine[11].

There are also some additional, in a way technical, challenges related to authentication and authorization. Many of the resources we model have their own security model and authorization schemes. Currently, the deployed model is embedded with authorization credentials (*e.g.* the endpoint `/nova-demo` uses OpenStack demo user and tenant).

B. Conclusion

This paper presents a novel use of *model@run.time* for infrastructure management. We have shown examples of using such model for managing OpenStack virtual machines implemented in an initial prototype. While preliminary and incomplete, it shows a great potential for system management, for building new higher-level abstractions on the top of it, and for its integration into existing self-adaptive engineering frameworks and tools.

We focused primarily on the infrastructure management, however, the same ideas can be applicable to any application that is manageable. For example, we can imagine a runtime model of a local email client[12], an iTunes library, or a desktop window manager. This might open a whole new window of opportunities for *model@run.time* research.

[11] http://bit.ly/1AMqrcJ

[12] Finding the email with a largest attachment will be then a simple query.

ACKNOWLEDGMENT

This work is partially financed by the Datalyse project[13].

[13] http://www.datalyse.fr

REFERENCES

[1] B. H. Cheng *et al.*, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," in *Software Engineering for Self-Adaptive Systems*, vol. 5525, 2009.

[2] N. Bencomo, R. France, B. H. Cheng, and U. Aßmann, *Models@run.time: Foundations, Applications, and Roadmaps*, 2014.

[3] G. Blair, N. Bencomo, and R. B. France, "Models@run.time", 2009.

[4] D. Garlan *et al.*, "Rainbow: architecture-based self-adaptation with reusable infrastructure," in *Computer*, 2004.

[5] F. Krikava, P. Collet, and R. Rouvoy, "Integrating Adaptation Mechanisms Using Control Theory Centric Architecture Models: A Case Study," in *International Conference on Autonomic Computing*, 2014.

[6] F. Krikava, P. Collet, and R. B. France, "ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures," in *Symposium on Applied Computing*, 2014.

[7] F. Krikava, P. Collet, and R. B. France, "SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations," in *International Conference on MODELS*, 2014

[8] T. Horn, "Model Querying with FunnyQT," in *International Conference on Model Transformation*, 2013

[9] X. Zhang *et al.*, "Demonstration of Runtime Model Based Management of Diverse Cloud Resources," in *Demo Session at MODELS'13*, 2013

[10] N. Huber, F. Brosig, and S. Kounev, "Modeling dynamic virtualized resource landscapes", in *International Conference on Quality of Software Architectures*, 2012

[11] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, Addison-Wesley Professional, 2008.

[12] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intelligence*, vol. 8, no. 3, pp. 323–364, Jun. 1977.

[13] B. Bennaceur, "Mechanisms for leveraging models at runtime in self-adaptive software," in *LNCS 8378*, 2014.

[14] T. Vogel and H. Giese, "Model-Driven Engineering of Self-Adaptive Software with EUREMA," in *ACM Transactions on Autonomous and Adaptive Systems*, Vol. 8, No. 4, Article 18

APPENDIX

— Get current CPU usage of all deployed VMs:

```
> nova.instances map (x => x.name -> x.status.cpu_usage)
| RTList((vm0,0.87), (vm1,0.27)))
```

— Get instance with the highest CPU usage:

```
> nova.instances maxBy (_.status.cpu_usage)
| NovaInstance(name="vm0", ...)
```

— Get the mean of CPU usage of all active VMs:

```
> nova.instances.filter(_.state==NovaState.Active)
                .map(_.status.cpu_usage).mean
| 0.57
```

— Spawn largest available VM with the latest Ubuntu:

```
> nova.instances += NovaInstance(name = "vm2",
   image=nova.images.filter (_.name startsWith "Ubuntu")
           .maxBy (_.name),
   flavor=nova.flavors.sorted.reverse
           .find (x => nova.quota isAvailable x))
```