ABS: a high-level modeling language for Cloud-Aware Programming *

Nikolaos Bezirgiannis, Frank de Boer

Centrum Wiskunde & Informatica (CWI), Amsterdam, Netherlands
{n.bezirgiannis, f.s.de.boer}@cwi.nl

Abstract. Cloud technology has become an invaluable tool to the IT business, because of its attractive economic model. Yet, from the programmers' perspective, the development of cloud applications remains a major challenge. In this paper we introduce a programming language that allows Cloud applications to monitor and control their own deployment. Our language originates from the Abstract Behavioral Specification (ABS) language: a high-level object-oriented language for modeling concurrent systems. We extend the ABS language with Deployment Components which abstract over Virtual Machines of the Cloud and which enable any ABS application to distribute itself among multiple Cloud-machines. ABS models are executed by transforming them to distributed-object Haskell code. As a result, we obtain a Cloud-aware programming language which supports a full development cycle including modeling, resource analysis and code generation.

1 Introduction

The IT industry, always looking for cutting operational costs, has been increasingly relying on virtualized resources offered by the "Cloud". Besides being more economically attractive, the Cloud can allow certain software to benefit in security and execution speed. For these reasons, software applications are steadily being migrated to run on virtualized hardware, essentially turning cloud computing into a hot topic among the software community.

Recent research has led to numerous methodologies, tools, and technologies being proposed to help the migration and execution of software in the cloud, ranging from (static) configuration management tools to (live) orchestration middleware, and from simple resource monitoring services to the dynamic (elastic) provisioning of resources. Unfortunately, the (so-called) DevOps engineers are now burdened with developing and maintaining an extra logic for such cloud tools, besides the usual application logic. These cloud tools may be best described as semi-automatic and it is often the case that an engineer has to manually intervene to apply the desired configuration&deployment of a cloud application.

These cloud applications are migrated unchanged: monolithic boxes of code which are transferred from a non-cloud setting to the new cloud environment by

* Partly funded by the EU project FP7-610582 Envisage. This work was carried out on the Dutch national e-infrastructure with the support of SURF Foundation.

the DevOps engineers. Such separation of the application from its execution is traditionally believed to be an advantage, long before Cloud came to existence. However, one would expect that with the introduction of the virtualized (dynamic) hardware of the Cloud, and since software logic is inherently dynamic, an application could "become aware" and leverage its own execution for managing its cloud resources & deployment in an optimal way, and without the constant administering of an engineer.

In this paper, we aim to address the challenges of engineering cloud applications by introducing a "cloud-aware" programming language that provides certain high-level abstractions for unifying the application logic together with its deployment logic in a single integrated environment, while in the same-time, hiding any lower-level hardware and cloud-provider considerations. The language is intended for DevOps engineers and (potentially) computational scientists who are responsible for both the development and execution of software residing in the Cloud but would rather focus more on the application's logic than manage continuously its deployment. Applications written in the proposed language are christened "cloud-aware" in the sense that they can *actively* monitor and control their own deployment.

The proposed language is based on the *ABstract Specification language* (ABS), a formally-specified, object-oriented modeling language that has been used for both analyzing [1], verifying ([8]), and simulating [5] software programs, as well as running them in production through the various backends developed (currently targeting Java, Erlang, and Haskell). We extend ABS with *Deployment Components* that serve as a suitable abstraction over Cloud Virtual Machines and which allow the application to distribute itself among multiple (provider-agnostic) computing systems. The ABS developer writes code that can dynamically create, monitor and shutdown such Deployment Components (Virtual Machines) and most importantly bring up new objects inside them. To this end, an ABS cloud-application forms a cloud-aware *distributed-object* system, which consists of a number of inter-VM objects that communicate asynchronously, while recording any failures that may happen in the cloud.

An implementation of this extension must be efficient and safe so that it can be put in production code. For this, the Haskell backend of ABS is chosen for translating ABS code to Haskell intermediate code, which is again typechecked and transformed to an executable by an external Haskell compiler. We augment this backend with support for *Cloud-Haskell*, a framework for type-safe, fault-tolerant distributed programming in the Haskell ecosystem. The implementation, although in its infancy, is already being tested in a real cloud environment, exhibiting promising results which are also presented.

2 ABS Language and its Cloud Extension

The ABS (for "ABstract Specification language")[5] is a statically-typed, executable modeling language with formal operational semantics. The language consists of a purely-functional programming core and an imperative, object-

oriented layer. The syntax and behaviour resembles that of Java with two clear differences: side-effectful code cannot be mixed with pure expressions, and class inheritance is abolished in favour of code reuse via delta models[3]. ABS adds, next to the Java-like (passive) objects, builtin support for active (concurrent) objects coupled with cooperative scheduling.

The *functional core* provides a declarative way to describe computation which abstracts from possible imperative implementations of data structures. The primitive types (Int and Rational) can be extended with (possibly recursive) algebraic data types (ADTs) (e.g. `data Bool = True | False`) that can exhibit parametric polymorphism (`List<A>`) and Hindley-Milner type inference. Pure expressions are formed by successive λ-`let` abstractions and applications over values of the defined datatypes (`let x = 3 in x>2 || True`). Function definitions associate a name to a pure expression which is evaluated in the scope where the the expression's free variables are bound to the function's arguments. The functional core supports pattern matching with a **case**-expression which matches a given expression against a list of branches.

The *imperative layer* specifies the interlaced control flow of the concurrent objects in terms of communication, synchronization, and internal computation. This layer extends the functional core (datatype and function definitions) with interface definitions, class definitions, and a main block. Interfaces declare a set of method names to their type-signatures. An interface `extends` other interfaces, in this case inheriting the methods of of its super-interfaces. A class definition declares its (private-only) attributes and a set of interfaces it `implements`. Method implementation bodies are comprised of statements of standard sequential composition $s; s$, assignment $x = rhs$, conditionals, while-loops, and return. Statements can mutate private attributes of the current class, locally-defined variables, and the method's formal parameters. The read-only variable **this** evaluates to the object in which computation occurs. A program's main block is a special method body with no *this* associated object. Classes are *not* types and used only to create object instances that instead are typed-by-interface. Note that interfaces support subtype polymorphism while ensuring strong encapsulation of implementation details.

Methods calls are either synchronous (`v = obj.method(args);`) where the statement is blocked until the method has finished with result v, or asynchronous (`f = obj!method(args);`) where the statement returns immediately with a *future* f (with type `Fut<A>`), without waiting for the method's completion. Each asynchronous method call creates a new *process* which will eventually store the result of the method call into the future reference. The caller can use this future reference to retrieve the result by calling the blocking statement `v = f.get;`. Objects may form a so-called *Concurrent Object Group* (COG), where objects (and their processes) share the same thread of control: at each point in time, only one process of the COG is executing. This process may decide to willfully pass control to another same-group process, by waiting until a future is ready (`await f?;`) or a boolean expression is met (`await exp;`). ABS does not specify

any concrete policy for this cooperative scheduling of processes; it is left to the particular implementation (backend) to decide.

2.1 Extending to the Cloud

We extend the ABS language with syntactic and library support for Deployment Components. A *Deployment Component* (DC), first described in [7], is "an abstraction from the number and speed of the physical processors available to the underlying ABS program by a notion of concurrent resource". Simply put, a DC corresponds to a single (properly-quantified) Virtual Machine which executes ABS code. We restrict the definition of DC to correspond only to a Platform Virtual Machine (VM) residing inside the boundaries of a Cloud infrastructure. Multiple inter-communicating VMs effectively form an ABS cloud application.

To be able to programmatically (at will) create and delete VMs in any language, would require modeling them as first-class citizens of that language. As such, we introduce DCs as first-class citizens to the already-existing language of ABS in the least-intrusive way: by modeling them as objects. All created DC objects are typed by the interface `DC`. Minimal implementation for the methods of the DC interface are `shutdown` for shutting down and releasing the cloud resources of a virtual machine, and `load` for probing its average *system load*, i.e. a metric for how busy the underlying computing-power stays in a period of time. We use the Unix-style convention of returning 3 average values of 1, 5 and 15 minutes. The DC interface resides in the augmented standard library:

```
module StandardLibrary.CloudAPI;
interface DC {
    Unit shutdown();
    Triple<Rat,Rat,Rat> load();
}
```

By having a common DC interface the different cloud backends can agree on a basic service, while still being able to provide additional functionality through sub-interfaces and distinct DC-interfaced classes. Each DC-interfaced class implements the connection to a distinct cloud provider (e.g. Amazon, Openstack). A code skeleton of such a class follows, where the DC (VM) is parameterized by the number of CPU cores and main RAM memory:

```
module StandardLibrary.SomeProvider;

data CpuSpec = Micro | Small | Large;
data MemSpec = GB(Int) | MB(Int);

class SomeProvider(CpuSpec c,MemSpec m) implements DC {
    Unit shutdown() { /*omitted*/ }
    Triple<Rat,Rat,Rat> load() { /*omitted*/ }
}
```

The implementor can expose other properties to DCs, such as, network, number of IO operations, VM region location. A concrete implementation, which is

omitted for brevity, usually involves some high-level ABS logic coupled with low-level code written in a foreign language (in our case Haskell). The average ABS user will not have to provide such connections to the cloud, since we (the implementors) intend to provide class implementations for most major cloud providers/technologies, in an accompanying ABS library. With this approach, we lift the low-level API of the cloud provider (usually XML-RPC) to a *typed* high-level API (e.g. CpuSpec and MemSpec datatypes).

Moving on, we create an object of the SomeProvider class by passing the number of cores and memory measured in GBs as class' formal parameters. The call to "new SomeProvider" contacts the specific cloud provider in the background for bringing up a new VM instance. The provider responds with a unique identifier (commonly the public IP address of the created VM) which is stored in the DC object. Finally, the machine is released by calling `shutdown()`, making the DC object point to `null`.

```
DC dc1 = new SomeProvider(Large, GB(8));
_ future_l1 = dc1 ! load();// underscore infers the type
_ l1 = future_l1.get;
dc1 ! shutdown();
```

The creation of a DC object reference is usually fast, since it involves a single network communication between the current ABS node and the cloud provider. Still, the underlying VM requires considerably more time to boot up and be responsive, depending on factors such as provider's availability, congestion and hardware. To address this, we allow the creation of new dc objects to continue, but we require the program to potentially block when executing the first operation of the newly-created DC, as shown in the example:

```
DC mail_server = new Amazon(..);
DC web_server = new Azure(..);
DC db_server = new Rackspace(..);
mail_server!load(); // will block if DC is not up yet
```

Similar to `this` identifier, a method context contains the **thisDC** read-only variable (with type DC) that points to the VM host of the current executing object. A running ABS node can thus control itself (or any other nodes), by getting its system load or shutting down its own machine. However, after its creation, a running ABS node will remain idle until some objects are created/assigned to it. The **spawns** keyword is added to create objects that "live" and execute in a remote DC:

```
Interf1 o1 = dc1 spawns Cls1(args..);
o1 ! method1(args..);
this.method2(o1);
```

The `spawns` behaves similar to the `new` keyword: it creates a new object (inside a new COG), initializes it, and optionally calls its run method. Indeed, the expression `new Cls1(params)` is equivalent to `thisDC spawns Cls1(params)`. The keyword spawns returns a *remote object reference*, (also called a proxy object; o1 in the above example) that can be called asynchronously for its methods

and passed around as a parameter. Every remote object reference is a single "address" *uniquely identified* across the whole network of nodes. Calls to `spawns` will also (besides `shutdown`, `load`) block a until the VM is up and running. From a theoretical standpoint, a remotely-spawned object must point to the same code (attributes and methods) as in a local object; a remark that is reinforced in the Subsection 3.1.

Whereas the development of ABS code is by-definition provider-dependent — the user has to explicitly specify the class of the cloud provider —, the communication and interaction between the spawned remote objects is (in principle) *provider-agnostic*. To this extent, an ABS user could write an ABS cloud application that spans over multiple cloud providers and, most importantly, different cloud technologies.

Cloud Failures In cloud computing, and in any distributed system in general, failures are more frequent, mostly because of unreliable networks. Based on this fact, we further extend ABS with proper support for extensible, asynchronous exceptions. At the language level, exceptions are pure expressions modeled as single-constructor values of the ADT `Exception`. To define new exceptions the user writes a declaration similar to an ADT declaration, e.g. `exception MyException(Int, List<String>);`. Our cloud extension predefines certain common "local" exceptions (e.g. NullPointerException, DivisionByZeroException) and cloud-related exceptions (e.g. NetworkErrorException, DCAllocationException, DecodingException).

Exception values are either implicitly raised by a primitive operation (e.g. DivisionByZeroException) or explicitly signaled using the `throw` keyword. To recover from exceptions the user writes a `try/catch/finally` block as in Java, the only difference being that the user can pattern-match on each catch-clause for the exception-constructor arguments. Normally, if an exception reaches the outermost caller without being handled, its process will stop. We introduce a special built-in keyword named `die` that changes this behaviour and causes an object to be nullified and *all* of its processes to stop. With this in hand, a distributed application can easily model objects that can be remotely killed:

```
interface Killable { Unit kill(); }
class K implements Killable {  Unit kill() { die; } }
Killable obj = dc1 spawns K();
obj ! kill();
```

Exceptions originating from asynchronous method calls are recorded in the future values and propagated to their callers. When a user calls "future.get;", an exception matching the exception of the callee-process will be raised. If on the other hand, the user does not call "future.get;", the exception will *not* be raised to the caller node. This design choice was a pragmatic one, to allow for fire-and-forget method calls versus method calls requiring confirmation. In our extension, we name this behaviour "lazy remote exceptions", analogous to lazy evaluation strategy.

3 Implementation

For the implementation, we rely on our abs2haskell backend/transcompiler. Haskell is a statically-typed, purely-functional language and, as such, it becomes straightforward to translate the ABS' functional core to Haskell. In the imperative layer, we model interfaces as Haskell's typeclasses, objects as references to mutable data (IORef in the Haskell world), and futures as synchronizing variables (MVar in Haskell). Nominal subtyping is achieved through an upcasting typeclass. An alternative would be to encode OO using extensible records [6], although this method widens the spectrum to structural subtyping.

At runtime, each COG becomes a Haskell lightweight thread (with SMP parallelism). The COG-thread holds a process-enabled *queue*, a process-disabled *table*, and a local *mailbox*. Upon an asynchronous method call, a new process is created and put in the end of the process-enabled queue; note that processes are not threads, they are coroutines (first-class continuations) and thus can be stored as data. The COG resumes the next process from the queue until it reaches an `await` (on a future or a condition), where the process is suspended and moved to the process-disabled table. Later, another process informs the COG (by writing to its mailbox) that the await-condition is met; the COG will move back the process to the enabled queue. This strategy avoids *busy-wait* polling the await conditions of processes.

Moving on to distributed programming, we extend our backend with layered support for Cloud-Haskell[4], a framework for Haskell that replicates Erlang's concurrency & distribution model (message passing) but in a type-safe manner. We reuse the network transports and serialization protocols defined in Cloud Haskell for the ABS transmitted data between Virtual Machines. Each COG-thread is accompanied with a separate Cloud-Haskell thread (also lightweight) that listens for messages in *public* mailbox and *forwards* them to the local mailbox of its associate COG-thread. This approach was chosen to firstly, avoid needless network-serialization between local communication and secondly, treat our distributed extension as optional to our (previously SMP-only) haskell backend.

Serialization ABS data have to be serialized to a standard format before transmitting them between DCs. The serialization of values of primitives and algebraic datatypes are automatically done by Haskell. We serialize object/future references to *proxy references* by serializing their Cloud-Haskell thread ID (network-unique) together with a COG-unique ID, and leaving out their actual attributes/future results. Each asynchronous method call is serialized to a *static closure*, i.e. a static code-pointer to the method (known at compile-time and platform-independent) and a serialized environment of its free variables (method arguments and local variables). No kind of code (source-, byte- or machine-code) corresponding to the method body is transferred. All described-above serializations are type-safe and version-safe, in the sense that we include next to the payload of an ABS datum, its serialized type signature and the library-versions of any types involved; thus, we avoid decoding bugs because of type and library-version mismatches.

Garbage Collection In a local-only setting, all ABS-based values, i.e. ADTs, futures, objects are automatically garbage-collected by the underlying Haskell GC. However, in our distributed setting some object/future references may have to be transmitted outside as proxy references, which results to the local ABS system garbage-collecting "too-early". An obvious solution would be to abolish automatic GC altogether, but that would hinder the development of software applications, especially those supposed to be long-running (as is the norm in cloud applications). On the other hand, introducing *distributed garbage collection* to ABS would allow both local and remote objects to be automatically GC'ed. The downside is that the user can no longer reason about the GC-incurred performance penalty which may be considerable. We chose a middleground where objects are by default GC-enabled and only become disabled when they are remotely communicated over (to another DC). The implementation has been straight-forward: a process appends the local object reference(s) that are transmitted remotely to a locally-held list of GC-disabled objects. This global list is held during the lifetime of the node, effectively surpassing the Haskell's garbage collector underneath. Our design choice was based on best practice; we believe that a distributed cloud ABS application of many DCs would contain a combination of a lot of local ephemeral objects, yet a few long-lived remote objects.

DCs, being special objects, are treated differently: when falling out of context they are automatically GC'ed. That does not mean that the attached VM is shut down. The user that wants to shutdown a DC but holds no reference to it anymore, has to contact a remote object residing there to return a reference to the DC (with `thisDC`), or to shut it down on user's behalf. If the executing program holds (now and in the future) no reference to a DC and its objects, we consider its VM unreachable and fallen out of scope of the ABS application.

Futures are garbage-collected in a publish-subscribe pattern: the caller of an asynchronous method is a subscriber, while the callee is the publisher. When the callee has finished computing the future, it "pushes" the resulted value to its caller (the direct subscriber) and may now locally garbage-collect that value. A subscriber that "passes over" a remote future reference to other nodes becomes an intermediate broker with the responsibility to later also "push" that future value to all others *before* it is allowed to locally garbage-collect it. This forwarding strategy avoids unnecessary tracking and network communication between the initial node and all (directly and indirectly) subscribed nodes.

Cloud Architecture When creating a new DC, a cloud provider is on the background contacted (usually via an XML-RPC API) and asked to bring up a new VM with the given characteristics. After the machine has booted, the caller replicates itself (the current ABS application) by transmitting its machine code to the newly-created machine. In case the cloud provider offers heterogeneous platforms (different OS or CPU architecture), we instead transmit the ABS source code and compile it in-place with our compiler toolset (that prior reside in the VM's image). The new machine runs the transmitted ABS application and sends an acknowledgment signal to its creator, which can now start computations to the new DC by spawning new objects in it.

When it comes to network communication between machines, Cloud-Haskell does not enforce any particular network transport; even different transports can be composed together. Some existing implementations are TCP, AMQP, CCI, in-memory, etc. In ABS, the particular transport used depends on the implementation of the DC-interfaced class: we currently have DC-class implementations for OpenNebula (TCP), Azure (TCP) and Local (in-memory).

4 Experimental Results

We tested two instances of a real-world load-balancer: one with a static deployment of workers, and an adaptive (dynamic) load-balancer with worker VMs created on-demand based on how "well" the workers can keep up with incoming requests. Clients were submitting job requests (of approximately of equal size) to the balancer in a steady rate; workers were distinct Cloud VMs that were continuously computing the results for their incoming job requests.

The *static* load-balancer case is a fairly straight-forward cloud ABS application, consisting of 3 classes of LoadBalancer, Worker, and Client, exchanging asynchronous method calls of job requests/results. The LoadBalancer runs the main block and initially creates N number of Worker DCs (VMs) before starting accepting requests and forwarding to workers in round-robin. We ran this static deployment against varying size (N=1..16) of worker VMs. The results of the runs are shown in Figure 1(a) stripped from the initial boot time of VMs. What we can draw from these results is that the completed jobs (per minute) nearly doubles when we double the number of worker VMs until we reach 5 workers. After that, we still increase the completed jobs but with a slower pace. This observation can be attributed to the fact that a point is reached where there is not a significant benefit from adding more worker VMs; the rate of job requests is always steady, thus worker VMs are "slacking".

We modified the static load-balancer to an *adaptive* version, that takes full advantage of the expressivivity of the cloud extension. The LoadBalancer creates now only 1 initial VM. We accommodate the LoadBalancer with a HeartBeater object which periodically retrieves the load of each worker in the VM "farm". The HeartBeater computes the average `load` of all VMs and if this average exceeds 80%, it creates a new DC (VM), adds it to the current farm, and remotely spawns a Worker in the new DC. We illustrate a particular run of this configuration in Figure 1(b) (NB: VM boot times are not subtracted from the result). Each asterisk $*$ in (b) is a point where the HeartBeater decides to create a new DC. This run stabilizes on 6 workers, which is a good approximation of maximum speed (according to Figure 1(a)), and possibly a good choice if we took into account any VM costs. As an extra, the HeartBeater could potentially `shutdown` machines if their load remained small (under a threshold) for a certain time.

The tests were conducted on the SURF cloud-provider with OpenNebula IaaS, modern 8-cores, each with 8GB RAM and 20Gbps Ethernet. Interesting to

mention is that each worker can benefit from ABS multicore (SMP) parallelism. A snippet of the HeartBeater follows with the full ABS code at our repository[1]:

```
class HeartBeater(List<Worker> farm, Balancer b) {
  Unit beat() {
    Rat avg = this.computeLoads(farm);
    if (avg > 80/100) {
      DC dc = new NebulaDC(8,8192); // 8-core, 8GB RAM
      Worker w = dc spawns Worker();
      farm = Cons(w,farm);
      b ! updateFarm(farm); } } }
```

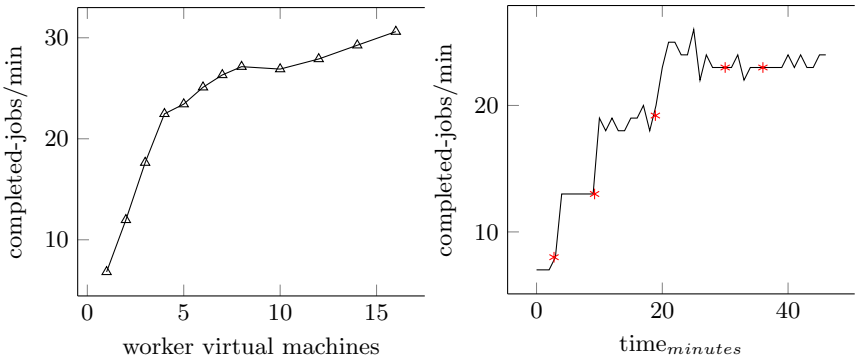

Fig. 1: (a) Static deployment of VMs (b) Adaptive Deployment over time

5 Related work

With the introduction of the Cloud, a plethora of cloud technologies & tools have appeared in the software community. We distinguish two categories of technologies related to our work: distributed-prog. languages and cloud middleware.

Distributed languages Erlang is one of the first distributed-oriented languages that next to the canonical message-passing communication, offers distinct features, such as hot-code loading and serialization of arbitrary closures. This comes with a cost in safety since the serialized Erlang data are untyped and usually unversioned. Erlang's builtin processes are lightweight threads whereas ABS processes are coroutines (even more lightweight). The Akka framework brings (typed) actors to the Scala language. Although Akka provides a rich library and toolkit, it currently lacks a cloud-aware API. At runtime Akka is constrained by a threadpool (since JVM threads are expensive) and actors are not able to use

[1] Upstream abs2haskell repository at http://github.com/bezirg/abs2haskell

cooperative scheduling and instead resort to a form of message routing. The Java RMI (Remote Method Invocation) is a library bundled in the Java platform for communication between remote objects. The product pioneered in areas such as bytecode downloading and distributed-GC. The method invocation is strictly synchronous (the caller has to wait for the remote method to finish) and thread-unsafe. JADE[2] is an active distributed multiagent system also built in Java; agents are more expressive than actors at the expense of program complexity and, possibly, performance.

Cloud middleware Ubuntu JuJu is a tool primarily for scaling and orchestrating a system's deployment on the cloud. Juju also comes with a GUI for modeling and visualizing a cloud deployment and saving it to a "recipe" for later reuse. It is usually accompanied by a configuration-management tool (such as Puppet) for the provisioning of cloud machines. CoreOS is a container-based OS that provides service and configuration discovery. It can be thought as a low-level infrastructure, primarily targeted to system administrators, for managing system services across a cluster of cloud machines, The Aeolus research project has built various tools that can derive an optimized deployment from the constraint-based model of a desired deployment, and automatically deploy that derivation. Finally, general SaaS supported by cloud providers eases the migration of existing software to the cloud and its automatic scaling of deployment. Albeit dynamic, a SaaS deployment can only vary on the CPU consumption, whereas our proposal would allow a much more expressive deployment that can depend on arbitrary application logic.

6 Conclusion and Future Work

We presented an extension to the ABS language that permits the management of an application's own cloud-deployment inside the language itself. We discussed the realization of such extension (by a Haskell transcompiler) and the execution of an ABS cloud application (based on Cloud-Haskell). Results showed that ABS can benefit from the extra performance that the Cloud offers. Moreover, the extension gives to ABS the expression power it needs to fuse the application logic with the application's own (dynamic) deployment logic. A positive side-effect of the proposed extension is that, ABS being primarily a modeling language, could now be used to model also an application's deployment. Indeed, such cloud-aware software models could be simulated against different and dynamically-varying cloud deployment scenarios.

For future work we are considering additions both at the language and run-time level. At the language level, it would be beneficial to include, besides the system load, other metrics such as memory, disk usage, object count, process count, exceptions raised. In this way, an ABS application would enhance its monitor and cloud-control logic. In a different direction, we plan to work on adding a basic *service discovery* mechanism to the standard library of ABS. This can be simply realized by extending the DC interface with two extra methods: an `acquire(Interface obj)` method that returns a reference to a remote

object implementing the provided Interface; an `expose(Interface obj)` that subscribes the passed object together with its current interface-view to the service registry of the DC.

At the system level, we are first interested in expanding our library support for other common cloud providers (such as Amazon EC2, OpenStack). Besides the current open (peer-to-peer) topology of DCs we want to add support for other cloud topologies, such as provider-specific, slave-master, or supervision topologies – a crude solution to topologies would be to introduce to the DC interface a method `List<DC> neighbours()` that lists all ABS nodes residing in the same private cloud network. A second consideration is to extend our virtualization technology support. With the introduction of micro-kernels (see the Xen hypervisor and unikernels), the cloud user no longer needs an OS underneath the application/service. By packaging the application into the kernel itself, the startup time of the VM is greatly improved, as is its management & distribution. The Haskell Lightweight Virtual Machine (HaLVM) is a promising such technology that allows the user to: "run Haskell programs without a host operating system". Likewise, *containers* (e.g. Docker), with its OS-level virtualization, would allow us to offer a more fine-grained control of deployment.

We believe that the cloud extension of ABS leads to new opportunities for furthering the application of formal methods to cloud computing, for example: specifying, verifying, and monitoring Service Level Agreements (SLA) of software systems — with that being the overall goal of ENVISAGE, our current research project. Indeed, we like to envisage software that is aware of its deployment and thus can control it, while its users merely monitor its behaviour via SLAs signed between the interested parties.

References

1. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: Costabs: a cost and termination analyzer for abs. In: PEPM. pp. 151–154 (2012)
2. Bellifemine, F., Poggi, A., Rimassa, G.: Jade–a fipa-compliant agent framework. In: Proceedings of PAAM. vol. 99, p. 33. London (1999)
3. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. ACM Sigplan Notices 46(2), 13–22 (2011)
4. Epstein, J., Black, A.P., Peyton-Jones, S.: Towards haskell in the cloud. In: ACM SIGPLAN Notices. vol. 46. ACM (2011)
5. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: Abs: A core language for abstract behavioral specification. In: FMCO. pp. 142–164 (2010)
6. Kiselyov, O., Lmmel, R., Schupke, K.: Strongly typed heterogeneous collections. In: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell. pp. 96–107 (2004)
7. Schäfer, J., Poetzsch-Heffter, A.: Jcobox: Generalizing active objects to concurrent components. In: ECOOP 2010–Object-Oriented Programming. Springer (2010)
8. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The abs tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. STTT 14(5), 567–588 (2012)