A Formal Model of Cloud-Deployed Software and its Application to Workflow Processing

Einar Broch Johnsen
Department of Informatics
University of Oslo
Oslo, Norway
einarj@ifi.uio.no

Ka I Pun
Department of Informatics
University of Oslo
Oslo, Norway
violet@ifi.uio.no

S. Lizeth Tapia Tarifa
Department of Informatics
University of Oslo
Oslo, Norway
sltarifa@ifi.uio.no

***Abstract*—Although insufficient software scalability and bad resource management can easily consume any potential savings from cloud deployment, the scalability of software executed on the cloud empowers the software designer, who can fully control the trade-offs between the incurred cost of running an application and the delivered quality-of-service. To capitalize on this control, the designer needs to be able to efficiently make deployment decisions in the software design, which are concerned with the chosen strategies for load balancing and scaling.**

By executable formal models and associated model-based analysis techniques, we explore and compare deployment strategies analytically in the laboratory during early design phases rather than empirically in the field after the code has been fully developed. We briefly survey ABS, a formal modeling language which integrates the design of services with the modeling of deployment decisions, and illustrate its use by an example of cloud-based workflow processing with autoscaling support.

I. Introduction

Scalable software deployed on the cloud can profit from the cloud infrastructure to meet work loads with the appropriate amount of resources. For regular work loads, this means that the amount of utilized resources can be fine-tuned to the particular work load. For irregular work loads, the elasticity of the underlying cloud infrastructure enables the software to dynamically vary the amount of resources it uses according to changes in the work load. Consequently, the scalability of software executed on the cloud empowers the software designer [1]: the designer of scalable applications can fully control the trade-offs between the incurred cost of running an application and the delivered quality-of-service.

Insufficient scalability and bad resource management for software services can easily consume any potential savings from cloud deployment [2]: Failed service-level agreements (SLAs) cause penalties for the provider, while oversized SLAs waste resources for the consumer. IBM Systems Sciences Institute estimates that a defect which costs one unit to fix in design, costs 15 units in testing (system/acceptance) and 100 units or more in production [3]; this cost estimation does not consider the *impact cost* due to, for example, delayed time to market, lost revenue, lost customers, and bad public relations.

Scalable software typically brings two ingredients to the design mix which are not traditionally present at the design stage in software engineering: strategies for *load balancing* and strategies for *scaling*. Whereas both can be bought as off-the-shelf solutions from cloud providers, many questions need to be answered before we can make good design decisions. Should load balancing and scaling be centralized as a monolithic orchestrator or decentralized into so-called *hot pools* [4]? Where are the bottlenecks if traffic increases? Off-the-shelf solutions do not always fit the bill if we are concerned with, e.g., fully exploiting pay-on-demand resources, enforcing company policies with respect to data protection, or optimizing resource usage based on business-level customer contracts rather than technology-level load measurements. As software goes from services to microservices and from virtual machines to containers, we can see a trend towards increasingly fine-grained and customized load balancing and scaling strategies.

The application of formal modeling and analysis techniques allows deployment decisions to be shifted from very late in the software engineering process to become an integral part of the software design. Using formal methods and model-based analyses, appropriate configurations and deployment decisions can be explored and compared "in the laboratory" [1], thereby helping users to predict the performance of an application before the application is deployed. For example, we have been doing simulations of 24 hour traffic patterns for different deployment choices of the HyVar tool-chain based on an executable formal model; each simulation took approximately five minutes on a standard laptop.

However, when our goal is to predict the behavior of deployed services by means of model-based analysis, it is important to distinguish aspects of the deployed services that we can fully analyze before deployment from factors that are outside our control. In our work on deployment decisions for cloud-deployed services, our approach has been to design and validate services by connecting executable models to formal service contracts and to an API that functions as a modeling abstraction of the cloud environment, see Fig. 1. This approach enables the following kinds of analysis:

- **Simulation ("Early modeling"):** Executable models let us use simulation tools for rapid prototyping and visualization of deployment decisions. To model deployment decisions, the formally defined modeling language ABS [5] realizes a separation of concerns between the *cost* of execution and the *capacity* of dynamically provisioned cloud resources [6].

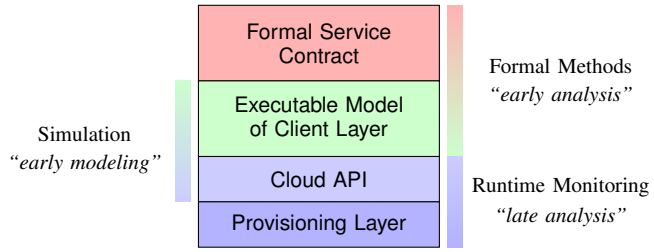

Figure 1. Predicting the behavior of cloud-deployed services by means of formal methods (figure from [1]).

- **Formal methods ("Early analysis"):** Because ABS was designed for analysis, it enables a range of tool-supported formal techniques, including type checking of behavioral types for deadlock analysis and SLA compliance [7], worst-case cost analysis with respect to different cost models such as processing and memory cost [8], deductive verification of behavioral properties, including protocol-like behavior [9], and automated test-case generation [10] with cutting edge partial order reduction techniques to eliminate redundant paths [11].
- **Monitoring ("Late analysis"):** Eventually, code can be generated from models by ABS backends [12] that preserve upper bounds on cost and permit performance monitoring of provisioned cloud resources after deployment [13]. We have used performance monitoring with respect to formal service contracts as a means to enable resource-aware services that scale according to their service-level agreements. For complex scaling, the ABS SmartDeployer [14], [15] uses constraint solving techniques to ensure that software components are correctly and optimally configured during the scaling process.

Successful applications of the modeling approach and analyses developed for ABS to cloud-deployed services in an industrial context include Fredhopper's eCommerce Platform [16] and Apache's Hadoop YARN [17].

To give a feel for how we model cloud-deployed software, we discuss the main modeling concepts of ABS in Sect. II and illustrate these by an example of an autoscaling application for processing user-defined workflows in Sect. III: this example used fine-grained container-like units of deployments.

II. Modeling Scalable Services in ABS

ABS is a modeling language for the development of executable models of distributed and deployed object-oriented systems with explicit deployment decisions. The main characteristics of ABS can be listed as follows:

1) it has a formal syntax and semantics;
2) it cleanly integrates concurrency and object orientation based on concurrent object groups (COGs) [5], [18];
3) it supports both synchronous and asynchronous communication [19], [20];
4) it offers a range of complementary modeling alternatives by integrating a functional layer with algebraic datatypes and functional programming, an imperative layer [5] with COGs and asynchronous communication, it allows the modeling of real-time behavior [21];
5) compared to object-oriented programming languages, it abstracts from low-level implementation choices for data structures, and compared to design-oriented languages like UML diagrams, it is executable and models the control flow of systems;
6) it supports *deployment modeling* by a separation of concerns between the resource costs of executions and the resource capacities of (virtual) locations. Deployment decisions can be made inside models [6], using a Cloud API to interact with the cloud provisioning layer [22];

The functional layer of ABS is used to model computations on the internal data of objects. It allows designers to abstract from the implementation details of imperative data structures at an early stage in the software design. The functional layer combines a language for parametric algebraic data types (ADTs) and a simple functional language with case distinction and pattern matching. ABS includes a library with predefined datatypes such as `Bool`, `Int`, `String`, `Rat`, `Unit`, etc. It also has parametric datatypes such as lists, sets and maps. All other types and functions are user-defined.

The imperative layer of ABS allows designers to express communication and synchronization between concurrent objects. In the imperative layer, processes are encapsulated within COGs [5], [18], the processes are created automatically at method call reception and terminated after the method call execution is finished. ABS combines active (with a `run` method which is automatically activated) and reactive behavior of objects. ABS is based on cooperative scheduling: Inside COGs processes may suspend at explicitly defined scheduling points, at which point control may be transferred to another process. Suspension allows other pending processes to be activated. However, the suspending process does not signal any particular process, instead the selection of the new process is left to the scheduler. In between these explicit scheduling points, only one process is active inside a COG, which means that race conditions are avoided. *Real Time ABS* [21] extends ABS with support for the modeling and manipulation of dense time. This extension allows to represent execution time inside methods. The local passage of time is expressed in terms of a statement called **`duration`** (as in, e.g., UPPAAL [23]). To express dense time, we consider a model represented by two types `Time` and `Duration`. Time values capture points in time as reflected on a global clock during execution. In contrast, finite durations reflect the passage of time.

The ABS Cloud API provides an interface to model cloud infrastructure in ABS [22]. The ABS Cloud API supports the dynamic acquisition and release of computing environments, such as virtual machines, which are modeled using *deployment components* [6]. A deployment component is a modeling abstraction which captures locations offering (restricted) resources to computations. The language also supports cost annotations to model resource consumption. The combination of deployment components with resource computations and cost annotations allows modeling implicit passage of time. In

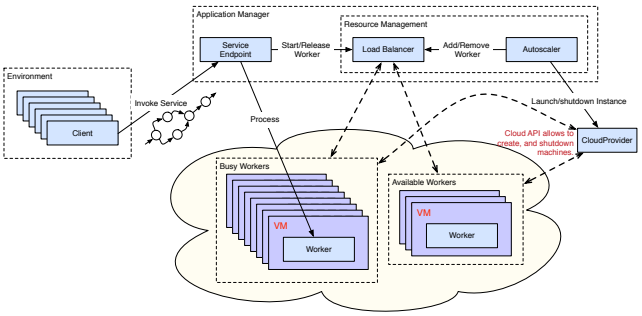

Figure 2. An architecture for a service with a resource-aware hot pool of workers which manage workflows of requests with dependencies.

this case, time can be observed by measuring the executing model, and monitoring the response time of a system. In this paper we will model so-called elastic computing resources, where the computation *speed* of virtual machines is determined by the amount of elastic computing resources allocated to these machines per discrete time interval (referred as *time interval* in the rest of the paper). *The ABS Cloud API* includes methods for launching and shutting down virtual machines. In the implementation, this is done by creating deployment components on which an application manager can deploy objects. In addition, the ABS Cloud API keeps track of the accumulated costs incurred by job execution on the cloud. ABS is supported by a range of analysis tools (see, e.g., [16]); for the analysis results in this paper, we are using the simulation tool which generates Erlang code.

III. Example

Let us consider a service, deployed on the cloud, which processes requests from clients in parallel. These requests contain user-defined workflows with inter-dependent tasks. The architecture of this service is shown in Fig. 2. The components of this service include a *service endpoint* which manages the workflows by requesting workers from a load balancer, and a *load balancer* which keeps a list of workers which are currently processing tasks and a list of workers which are available for new incoming tasks, and distributes the tasks evenly among the workers. Each *worker* can process a task in the workflow, and for that it waits for all task dependencies to be finished before it can proceed with its own execution. The service also has an *autoscaler* which will increase or decrease the number of workers depending on the workload. For this example, we will deploy the workers on the cloud, where each worker has its own dedicated virtual environment and we can abstract from the deployment outside the cloud; therefore, the *application manager* is not explicitly deployed in our model. We here assume that deploying workers is a lightweight operation, reminiscent of container technologies rather than heavyweight virtual machines, but the modeling language is not restricted to this assumption (i.e., it is easy to model delays associated to deployment operations).

Figure 2 shows a number of clients calling the service concurrently. Each *client* makes a number of requests, each of

```
type TaskID = Int;
type WFlow = Set<Task>;
data Task = Task(TaskID taskID, Rat cost,
                 Set<TaskID> dependencies);
// Example of a workflow
WFlow wf = set[Task(1,70,set[ ]),Task(2,85,set[1]),
              Task(3,60,set[1]),Task(4,73,set[2,3]),
              Task(5,81,set[4])];
```

Figure 3. Data types for modelling workflows and tasks in ABS.

which consists of a workflow with tasks. In this example, we combine the behavior of two different kinds of clients, open and closed, to create various patterns in client traffic. A *closed client* behaves as follows: in each iteration, the client waits for its cycle to finish before it sends the next invocation of the service and waits for the result. Consequently, a closed client will not flood the service. In contrast, an *open client* behaves as follows: in each cycle this client sends an invocation without waiting for the result. If the cycle is very short, the service may receive a sudden burst of requests, potentially flooding the service. Both types of clients will finish their execution once they have sent the desired number of requests.

This example is a variation of the example from [2], which only handles requests with single tasks. We now focus on how to model workflows and on a service for workflow processing with a simple autoscaler. The model is parametric in autoscaling strategy in the sense that the autoscaler is encapsulated behind an interface and can be replaced. Further details about the remaining components can be found in [2].[1]

A. Modeling and Deploying Workflows

To model workflows and tasks, we define ABS data types, shown in Fig. 3. A workflow `WFlow` is defined as a set of tasks. A task `Task` has an identifier, a constant processing cost, and a set of task dependencies. Consider for example the workflow (shown below) with five tasks. The workflow starts with Task `1`, after this task is executed, Tasks `2` and `3` can start executing in parallel. When both tasks have completed, Task `4` can start to execute. When it has finished, Task `5` starts its execution to complete the workflow.

Figure 4 illustrates the `ServiceEndpoint` model. The method `invokeService` records the time when the workflow started its execution, and invokes a private method `manWF` to manage the distribution of the workflow tasks to different workers. Note that the workflow has a deadline, so a request is considered successful if the workflow execution meets this deadline. The method `manWF` keeps a map associating task identifiers for tasks already sent to workers to corresponding *futures* **`Fut`**`<Bool>`, where the replies from the workers will eventually be stored. The first part of the model recurses through the workflow's tasks until all tasks have been sent to different workers for execution. The function `chooseTs` chooses a task in the workflow whose task dependencies have already sent for execution, so the associated future identifiers

[1]The full example can be retrieved from github (http://goo.gl/ZxUzH6). ABS can be tried in the online collaboratory [24] (http://abs-models.org).

```
interface SE {
  Bool invokeService(WFlow wf, Duration deadline); }

class ServiceEndpoint(LoadBalancer lb) implements SE {
 Bool invokeService(WFlow wf,Duration deadline){
  Time started = now();
  Bool success=await this!manWF(started,wf,deadline);
  return success;}

 Bool manWF(Time started,WFlow wf,Duration deadline){
  Bool success = True;
  Map<TaskID,Fut<Bool>> replies = map[ ];
  while (!emptySet(wf)) {
   Maybe<Task> maybeTs= chooseTs(wf, keys(replies));
   case maybeTs {
    Nothing => println("Circular dependencies");
    Just(tsk) =>{
     Worker w = await lb!getWorker();
     Fut<Bool> reply=
         w!processTask(cost(tsk),started,deadline,
           filterFut(dependencies(tsk),replies));
     replies=insert(replies,Pair(taskID(tsk),reply));
     wf = remove(wf,tsk);
     } } }
  while(!emptyMap(replies)) {
    TaskID tid =  take(keys(replies));
    Fut<Bool> reply = lookup(replies, tid);
    await reply?;
    Bool result = reply.get;
    success = (result && success);
    replies = removeKey(replies,tid);}
  return success;}
  ...
```

Figure 4. The service endpoint which manages workflows in ABS (extract).

```
def Maybe<Task> chooseTs(WFlow wf, Set<TaskID> tids)=
   if emptySet(wf)
   then Nothing
   else
    let (Task h) = take(wf) in
    if isSubset(dependencies(h),tids)
    then Just(h)
    else chooseTs(remove(wf, h), tids);

def List<Fut<Bool>> filterFut(Set<TaskID> tids,
                    Map<TaskID,Fut<Bool>> replies) =
   if emptySet(tids)
   then Nil
   else let (TaskID h) = take(tids) in
      appendright(filterFut(remove(tids, h),replies),
                  lookup(replies, h));
```

Figure 5. Functions to model task selection from workflows in ABS.

```
class WorkerObject(LoadBalancer lb) implements Worker {
  Bool processTask(Rat taskCost, Time started,
     Duration deadline, List<Fut<Bool>> dependencies){
    while(dependencies != Nil) {
      Fut<Bool> dep =  head(dependencies);
      dependencies = tail(dependencies);
      await dep?;}
    [Cost: taskCost] skip;
    Rat spentTime = timeDifference(now(),started);
    lb!releaseWorker(this);
    return (spentTime <= durationValue(deadline));}
  ...
```

Figure 6. A model of a worker in ABS (extract).

```
class Autoscaler(CloudProvider cloud, LoadBalancer lb,
        Int nbrOfWorkers, Int nResources,
        Int cycle, Counter c) implements Autoscaler {
  Unit run(){
    Int ctr = 0;
    while (ctr<nbrOfWorkers) {
      Fut<DC> fs =  cloud!launchInstance(
                    map[Pair(Speed, nResources)]);
      DC vm = fs.get;
      [DC: vm] Worker w = new WorkerObject(lb);
      lb!addWorker(w);
      Time startTime = await vm!getCreationTime();
      await c!addMachine(vm,startTime);
      ctr=ctr+1; }
    this!resize();
  }
  Unit resize(){
    Int ctr = 0;
    await duration(cycle, cycle);
    Int available = await lb!getNbrAvailableW();
    Int inuse = await lb!getNbrInuseW();

    if (...) { // Threshold for scaling up
      Rat extraworkers= ... // Number of new workers
      while (ctr<extraworkers) {
        ... }} // Add workers as in the run method
    if (...) { // Threshold for scaling up
      Rat removeworkers= available/2;
      while (ctr<removeworkers) {
        Worker w = await lb!removeWorker();
        DC dc = await w!getDC();
        Bool down = await cloud!shutdownInstance(dc);
        await  c!addShutdown(dc, now());
        ctr=ctr+1;
    } }
    this!resize();
  } }
```

Figure 7. A template for an autoscaling strategy in ABS (extract).

are in the map `replies`. We here assume that workflows are well-formed, so all tasks can eventually be selected. Once a task is chosen, we call the load balancer `lb` to get a worker, and send the task to this worker, together with its cost, the starting time and deadline of the workflow, and the futures associated with the task dependencies, extracted with a function `filterFut`. The functions `chooseTs` and `filterFut` are defined in Fig. 5. After all tasks have been sent to different workers, we start collecting all the replies and check if each of the tasks manages to finish its execution within the deadline of the whole workflow.

The workers processing tasks can be modeled by a class `WorkerObject`, illustrated in Figure 6. The method `process` abstracts from the functional behavior by a cost annotation, and captures that the task can only execute once its task dependencies have finished. The model can be refined with explicit functionality and fine-grained cost annotations; e.g., costs can depend on output data from previous tasks, which can be modeled in ABS using expressions in cost annotations. Finally, the method checks if it has kept its deadline. Since we focus on QoS, we model soft deadlines.

B. Deployment Decisions & Their Visualization

The autoscaler, shown in Fig. 7, initially creates a given number `nWorkers` of workers in its `run` method by interacting with the cloud provider (which implements the Cloud API abstraction, see Fig. 1); these workers are deployed on ABS deployment components with `nResources` each. As the model executes the autoscaler's `resize` method will create and destroy workers according to a scaling strategy which depends on the workload. Scheduling decisions are made every `cycle` time intervals. The counter `c` is simply a monitor which stores a time series of allocated deployment components. New workers are handed over to the load bal-

```
{     ...
    CloudProvider cloud = new CloudProvider("cloud");
    LoadBalancer lb = new RoundRobinLoadBalancer();
    Autoscaler as = new Autoscaler(cloud,lb,nWorkers,
                       nResources,interval);
    SE endpoint = new ServiceEndpoint(lb);
    // Start clients to generate a desired workload
    ...
}
```

Figure 8. Dynamic deployment of the workflow processing service.

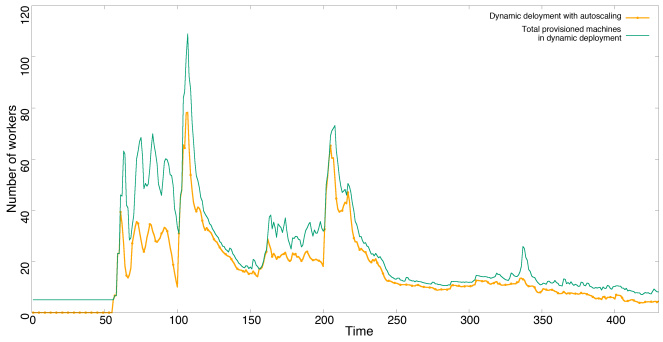

Figure 9. Average provisioning of virtual machines and machines in use for the deployment scenario of the example.

ancer `lb`, who keep track of available and busy workers and distributes tasks following a round-robin strategy.

We initiate our system, as shown in Fig. 2, in the model's main block (Fig. 8) by creating instances of the cloud provider (predefined in ABS), the service endpoint, the load balancer, and the autoscaler. We also need a workload describing some desired behavioral patterns in terms of open and closed clients with workflow requests.

We can immediately observe the behavior of our service with this deployment model by means of simulations. Such early analysis results at design time allows us to observe how the model complies with non-functional properties related to quality of service and build scalability into the design. We run this scenario 100 times using the ABS simulator. In each run we record the total number of workers and the number of busy workers which have been in use in each time interval.

Figure 9 captures the number of provisioned containers (total number of workers) and the number of containers in use (workers in use) per time interval for the scenario of the example, averaged over 100 simulations. The green and orange lines capture the varying number of provisioned containers and of containers in use over a period of 430 time intervals, respectively. In the simulation of this scenario, the total number of provisioned containers and the number of containers in use vary according to the workload, while the periods of both over-provisioning of containers and with congestion are rather short. Observe that workload peaks occur when open clients are operating.

IV. Related Work

The main focus of general-purpose modeling languages has been on abstractions for describing functional behavior and logical composition. However, this is inadequate for virtualized systems such as clouds when the software's deployment influences its behavior and when virtual processors are dynamically created. A large body of work on performance analysis for embedded systems using formal models can be found based on, e.g., process algebra [25], Petri Nets [26], and timed and probabilistic automata [27], [28]. These works mainly focus on non-functional aspects of embedded systems without associating capacities with locations. A more closely related technique for modeling deployment can be found in an extension of VDM++ for embedded real-time systems [29], in which static architectures are explicitly modeled using buses and CPUs with fixed resources. A recent, axiomatic approach in Event-B uses refinement to introduce elasticity in components [30]. Compared to these languages, Real-time ABS [6], [21] provides a formal basis for modeling not only timed behavior but also dynamically created resource-constrained deployment architectures, which enables users to model feature-rich object-oriented distributed systems with explicit resource management at an abstract and precise level.

Related work on simulation tools for cloud computing is mostly reminiscent of network simulators. Testing techniques and tools for cloud-based software systems are surveyed in [31]. In particular, CloudSim [32] and ICanCloud [33] are simulation tools using virtual machines to simulate provisioning in cloud environments. CloudSim is a mature tool which has been used for a number of papers, but it is restricted to simulations on a single computer. In contrast, ICanCloud supports distribution on a cluster. EMUSIM [34] is an integrated tool that uses AEF (Automated Emulation Framework) to estimate performance and costs for an application by means of emulations to produce improved input parameters for simulations in CloudSim. Compared to these approaches, our work aims to support the developer of client applications for cloud-based environments at an early phase in the software engineering process and is based on a formal semantics.

V. Conclusion

In this paper, we advocate the use of executable formal models and associated model-based analysis techniques in the engineering of scalable software targeting cloud deployment. Executable formal models have a low entry point as illustrated by the example in this paper: ABS has a syntax familiar to programmers and is easy to use in "early modeling" of cloud deployment strategies for, e.g., load balancing and scaling. Whereas simulations as used in this paper only cover single runs of a system, formal methods enable us to analyze worst-case situations. Going beyond the example in this paper, ABS provides support for automated deadlock analysis, for worst-case resource analysis to derive cost expressions such as those used in our workflows from code, automated test-case generation, and formal verification of functional correctness. These analysis techniques are enabled by the formal semantics of the modeling language. Industrial usage of ABS includes the development of SLA-based monitoring and scaling strategies, where the functional correctness of the monitoring and scaling

code was verified. Our experiences here indicate model-based decision making about deployment strategies enable higher quality and more cost-efficient services, both as part of the initial development process and as a tool for DevOps teams.

References

[1] R. Hähnle and E. B. Johnsen, "Designing resource-aware cloud applications," *IEEE Computer*, vol. 48, no. 6, pp. 72–75, 2015.

[2] E. B. Johnsen, K. I. Pun, and S. L. Tapia Tarifa, "Modeling deployment decisions for elastic services with ABS," in Proc. First Intl. Workshop on *Formal Methods for and on the Cloud*, ser. Electronic Proceedings in Theoretical Computer Science, vol. 228. Open Publishing Association, 2016, pp. 16–26.

[3] B. W. Boehm and P. N. Papaccio, "Understanding and controlling software costs," *IEEE Trans. SW Eng.*, vol. 14, no. 10, pp. 1462–1477, 1988.

[4] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[5] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, "ABS: A core language for abstract behavioral specification," in *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, ser. Lecture Notes in Computer Science, vol. 6957. Springer, 2011, pp. 142–164.

[6] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa, "Integrating deployment architectures and resource consumption in timed object-oriented models," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 67–91, 2015.

[7] E. Giachino, C. Laneve, and M. Lienhardt, "A framework for deadlock detection in core ABS," *Software and System Modeling*, vol. 15, no. 4, pp. 1013–1048, 2016.

[8] E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez, "SACO: Static Analyzer for Concurrent Objects," in *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, ser. Lecture Notes in Computer Science, vol. 8413. Springer, 2014, pp. 562–567.

[9] C. C. Din, R. Bubel, and R. Hähnle, "KeY-ABS: A deductive verification tool for the concurrent modelling language ABS," in *Proceedings of the 25th International Conference on Automated Deduction (CADE 2015)*, ser. Lecture Notes in Computer Science, vol. 9195. Springer, 2015, pp. 517–526.

[10] E. Albert, M. Gómez-Zamalloa, and M. Isabel, "SYCO: A systematic testing tool for concurrent objects," in *25th International Conference on Compiler Construction (CC'16)*. ACM, 2016, pp. 269–270.

[11] E. Albert, P. Arenas, M. G. de la Banda, M. Gomez-Zamalloa, and P. Stuckey, "Context-sensitive dynamic partial order reduction," in *Proc. 28th International Conference on Computer Aided Verification (CAV 2017)*, 2017, to appear.

[12] N. Bezirgiannis and F. de Boer, "ABS: A high-level modeling language for cloud-aware programming," in *SOFSEM 2016: Theory and Practice of Computer Science*. Springer, 2016, pp. 433–444.

[13] B. Nobakht, S. de Gouw, and F. S. de Boer, "Formal verification of service level agreements through distributed monitoring," in *Proceedings 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*, ser. Lecture Notes in Computer Science, vol. 9306. Springer, 2015, pp. 125–140.

[14] S. de Gouw, M. Lienhardt, J. Mauro, B. Nobakht, and G. Zavattaro, "On the integration of automatic deployment into the ABS modeling language," in *Proceedings 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*, ser. Lecture Notes in Computer Science, vol. 9306. Springer, 2015, pp. 49–64.

[15] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, and J. Mauro, "Zephyrus2: On the fly deployment optimization using SMT and CP technologies," in *Proc. 2nd International Symposium on Theories, Tools, and Applications (SETTA 2016)*, ser. Lecture Notes in Computer Science, vol. 9984. Springer, 2016, pp. 229–245.

[16] E. Albert, F. S. de Boer, R. Hähnle, E. B. Johnsen, R. Schlatte, S. L. Tapia Tarifa, and P. Y. H. Wong, "Formal modeling of resource management for cloud architectures: An industrial case study using Real-Time ABS," *Journal of Service-Oriented Computing and Applications*, vol. 8, no. 4, pp. 323–339, 2014.

[17] J.-C. Lin, I. C. Yu, E. B. Johnsen, and M.-C. Lee, "ABS-YARN: A formal framework for modeling Hadoop YARN clusters," in *19th International Conference on Fundamental Approaches to Software Engineering (FASE 2016)*, ser. Lecture Notes in Computer Science, vol. 9633. Springer, 2016.

[18] J. Schäfer and A. Poetzsch-Heffter, "JCoBox: Generalizing active objects to concurrent components," in *European Conference on Object-Oriented Programming (ECOOP 2010)*, ser. Lecture Notes in Computer Science, vol. 6183. Springer, Jun. 2010, pp. 275–299.

[19] E. B. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," *Software and Systems Modeling*, vol. 6, no. 1, pp. 35–58, Mar. 2007.

[20] F. S. de Boer, D. Clarke, and E. B. Johnsen, "A complete guide to the future," in *Proc. 16th European Symposium on Programming (ESOP'07)*, ser. Lecture Notes in Computer Science, R. de Nicola, Ed., vol. 4421. Springer, Mar. 2007, pp. 316–330.

[21] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa, "User-defined schedulers for real-time concurrent objects," *Innovations in Systems and Software Engineering*, vol. 9, no. 1, pp. 29–43, 2013.

[22] E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa, "Modeling resource-aware virtualized applications for the cloud in Real-Time ABS," in *Proc. Formal Engineering Methods (ICFEM'12)*, ser. Lecture Notes in Computer Science, vol. 7635. Springer, Nov. 2012, pp. 71–86.

[23] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, 1997.

[24] J. Doménech, S. Genaim, E. B. Johnsen, and R. Schlatte, "EasyInterface: A toolkit for rapid development of guis for research prototype tools," in *Proc. 20th International Conference on Fundamental Approaches to Software Engineering (FASE 2017)*, ser. Lecture Notes in Computer Science, vol. 10202. Springer, 2017, pp. 379–383.

[25] F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone, "Space-aware ambients and processes," *Theoretical Computer Science*, vol. 373, no. 1–2, pp. 41–69, 2007.

[26] M. Sgroi, L. Lavagno, Y. Watanabe, and A. Sangiovanni-Vincentelli, "Synthesis of embedded software using free-choice Petri nets," in *Proc. 36th ACM/IEEE Design Automation Conference (DAC'99)*. ACM, 1999, pp. 805–810.

[27] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "TIMES: a tool for schedulability analysis and code generation of real-time systems," in *Proc. FORMATS'03*, ser. Lecture Notes in Computer Science, vol. 2791. Springer, 2003, pp. 60–72.

[28] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen, "Performance evaluation and model checking join forces," *Comm. ACM*, vol. 53, no. 9, pp. 76–85, 2010.

[29] M. Verhoef, P. G. Larsen, and J. Hooman, "Modeling and validating distributed embedded real-time systems with VDM++," in *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*, ser. Lecture Notes in Computer Science, vol. 4085. Springer, 2006, pp. 147–162.

[30] M. Graiet, L. Hamel, A. Mammar, and S. Tata, "A verification and deployment approach for elastic component-based applications," *Formal Aspects of Computing*, Mar 2017.

[31] X. Bai, M. Li, B. Chen, W.-T. Tsai, and J. Gao, "Cloud testing tools," in *Proc. 6th Intl. Symposium on Service Oriented System Engineering (SOSE'11)*. IEEE, 2011, pp. 1–12.

[32] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. D. Rose, and R. Buyya, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software, Practice and Experience*, vol. 41, no. 1, pp. 23–50, 2011.

[33] A. Nuñez, J. Vázquez-Poletti, A. Caminero, G. Castañé, J. Carretero, and I. Llorente, "iCanCloud: A flexible and scalable cloud infrastructure simulator," *Journal of Grid Computing*, vol. 10, pp. 185–209, 2012.

[34] R. N. Calheiros, M. A. S. Netto, C. A. F. D. Rose, and R. Buyya, "EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications," *Software: Practice and Experience*, vol. 43, no. 5, pp. 595–612, 2013.