Automated Synthesis and Deployment of Cloud Applications *

Roberto Di Cosmo
roberto@dicosmo.org

Michael Lienhardt
michael.lienhardt@inria.fr

Ralf Treinen
treinen@pps.univ-paris-diderot.fr

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Jakub Zwolakowski
jakub.zwolakowski@inria.fr

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France

Antoine Eiche
Mandriva, FR
aeiche@mandriva.com

Alexis Agahi
Kyriba Corporation, USA
alexis.agahi@kyriba.com

ABSTRACT

Complex networked applications are assembled by connecting software components distributed across multiple machines. Building and deploying such systems is a challenging problem which requires a significant amount of expertise: the system architect must ensure that all component dependencies are satisfied, avoid conflicting components, and add the right amount of component replicas to account for quality of service and fault-tolerance. In a cloud environment, one also needs to minimize the virtual resources provisioned upfront, to reduce the cost of operation. Once the full architecture is designed, it is necessary to correctly orchestrate the deployment phase, to ensure all components are started and connected in the right order.

We present a toolchain that automates the assembly and deployment of such complex distributed applications. Given as input a high-level specification of the desired system, the set of available components together with their requirements, and the maximal amount of virtual resources to be committed, it synthesizes the full architecture of the system, placing components in an optimal manner using the minimal number of available machines, and automatically deploys the complete system in a cloud environment.

*This work was supported by the French ANR project ANR-2010-SEGI-013-01 Aeolus and partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, `http://www.irill.org`

ASE'14, September 15-19, 2014, Vasteras, Sweden.

ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642980.

1. INTRODUCTION

In contrast to classic, monolithic software that runs locally on a single machine, large distributed systems are built from many *running services* executing on (possibly heterogeneous) *virtual machines* (or *locations*) and collaborating to provide the expected functionality to final users.

The system *architect* must choose which services to use and how to configure them, knowing that services may depend on, and/or be in conflict with, each other; consider fault tolerance and quality of service issues, and provide enough instances of each service; design the physical architecture on which to run the system, trying to keep its *cost* reasonable with nonetheless enough locations with enough *resources* (e.g. RAM, disk space, bandwidth) to allow the installation and the good execution of the services they host; choose which implementation of each service to install on which location, knowing that implementations (usually in the form of *packages*) have dependencies and conflicts too. Once all this planning is done, the *deployment* phase must provision the required virtual machines, install the right packages on each of them, and finally start and interconnect services in the right order.

This is a daunting task, not unlike building a puzzle—each running service, package, and machine being a piece—where one only knows the overall expected functionality.

To reduce the complexity of this process, many industrial initiatives develop tools [32, 5] that allow to select, configure, and push on the Cloud some well defined services. However, these tools are only useful once the puzzle is solved, i.e. when the right services and packages have been selected, the locations on which they must be deployed have been chosen, and the way of configuring them in a manner that satisfies all the requirements has been found. Solving this puzzle currently requires a significant amount of human expertise, so that in practice large software stacks are often managed using custom scripts and manual techniques, which are error-prone and fragile [23].

In this article, we present a toolchain developed in the framework of the Aeolus project [6] that provides a generic, automatic and sound alternative to these scripts and techniques. The toolchain is composed of two tools:

`Zephyrus` automatically generates an abstract representation (or *configuration*) of the target system according to a concise specification of the expected functionalities. It takes into account the set of available services, which can serve as building blocks, with their requirements, replication policies, and resource consumption characteristics; information concerning the implementation of the services (e.g. that the `Apache` service is provided by the `www-servers/apache` package on `Gentoo` Linux, by the `apache2` package on `Debian`, etc.); and the maximum amount of (virtual) machines available, together with their characteristics. It is also able to minimize the amount of needed resources.

`Armonic` takes the full system configuration produced by `Zephyrus` and deploys it, by provisioning the required virtual machines on a cloud computing platform (such as OpenStack), installing the needed packages on each machine; it configures the different services to establish the required connections; and finally starts the services in the right order, relying on precise metadata that describe the internal state machines and runtime dependencies of each service.

This toolchain decouples system design from system deployment. It builds upon the sound formal foundations of the Aeolus component model [9], which describes each available service as a *component type*, using *ports* tagged with an *arity* to encode requirements, provides, conflicts and replication policy, as well as an internal state machine to capture component life-cycles.

`Zephyrus` uses a stateless version of the Aeolus model, extended to take into account locations, repositories, packages, and resources, as detailed in [7]; packages and repositories are encoded following the now standard approach originated from [21]. The specifications accepted by `Zephyrus` are given in a rigorous syntax whose semantics defines when a configuration satisfies a specification. Based on this formalization, `Zephyrus` can be proven correct and complete: it will always find a configuration that is optimal w.r.t. the chosen criterion if one exists. Furthermore, the generated configuration is guaranteed to provide the expected functionalities, and satisfies the constraints defined by the replication policies, as well as the dependencies and conflicts between services.

`Armonic` then takes over and uses the information about the internal state machine of the components to determine a correct service activation sequence, under the assumption that the dependency relation among services is acyclic, which is usually the case.

Paper structure. Section 2 presents the toolchain through a realistic running example. Section 3 discusses the internals of the various tools and presents the theoretical results establishing soundness, completeness and complexity of the architecture synthesis phase. Section 4 reports on experimentation with the toolchain, as well as its adoption in an industrial context at Kyriba Corporation. Before concluding, Section 5 discusses related work.

2. WALKTHROUGH

In this section we describe `Zephyrus` and `Armonic` by showing them at work on an example that is simple enough to be fully presented, and yet realistic as it corresponds to a common use case of application deployment in the cloud.

`Zephyrus` takes several inputs:

1. a description of all the existing components and their constraints, which come in various formats due to their different origins (e.g. package database, architectural choices, machine physical resources, etc.); this is called a *universe*.
2. a description of the *current system configuration* (existing machines, which services are currently deployed where, etc.)
3. a high level *specification* of the desired system. As part of the specification, architects can include objective functions that they would like to optimize for, such as the desire of minimizing the number of virtual machines that will be used for the deployment (and hence the system cost).

2.1 Use case: deploying a WordPress farm

The task we want to perform is deploying the popular Wordpress blog platform on a private OpenStack cloud. In addition to being realistic, this use case is often used as a "benchmark" to showcase the characteristics of cloud provisioning platforms. Wordpress is written in PHP and as such is executed within Web server software like Apache or nginx. Additionally, Wordpress needs a connection to a MySQL instance, in order to store user data. Simple Wordpress deployments can therefore be obtained on a single machine where both Wordpress and MySQL get installed.

"Serious" Wordpress deployments, however—that sustain high load and are fault tolerant—are more complex and rely on some form of load balancing. One possibility is to balance load at the DNS level using servers like Bind: multiple DNS requests to resolve the website name will result in different IPs from a given pool of machines, on each of which a separate Wordpress instance is running. Alternatively one can use as website entry point an HTTP reverse proxy capable of load balancing (and caching, for added benefit) such as Varnish. Either way, Wordpress instances will need to be configured to contact the same MySQL database, to avoid delivering inconsistent results to users. Also, having redundancy and balancing at the front-end level, one usually expects to have them also at the DBMS level. One way to achieve that is to use a MySQL *cluster*, and configure the Wordpress instances with multiple entry points to it.

Constraints. Several design constraints should be taken into account when designing such a system. Some constraints come from package providers and cannot be easily changed. For instance, Wordpress, Varnish, etc. usually come from software distribution packages and have their own dependencies and conflicts which must be respected on each machine when installing the software.

On the other hand, "house" requirements are defined by system architects to capture some ad-hoc policy. For this use case, we assume given the following requirements:

- at least 3 replicas of Wordpress behind Varnish or, alternatively, at least 7 replicas with DNS-based load balancing (since DNS-based load balancing is not capable of caching, the expected load on individual Wordpress instances is higher);
- at least 2 different entry points to the MySQL cluster;
- each MySQL instance shouldn't serve the needs of more than 3 Wordpress instances;

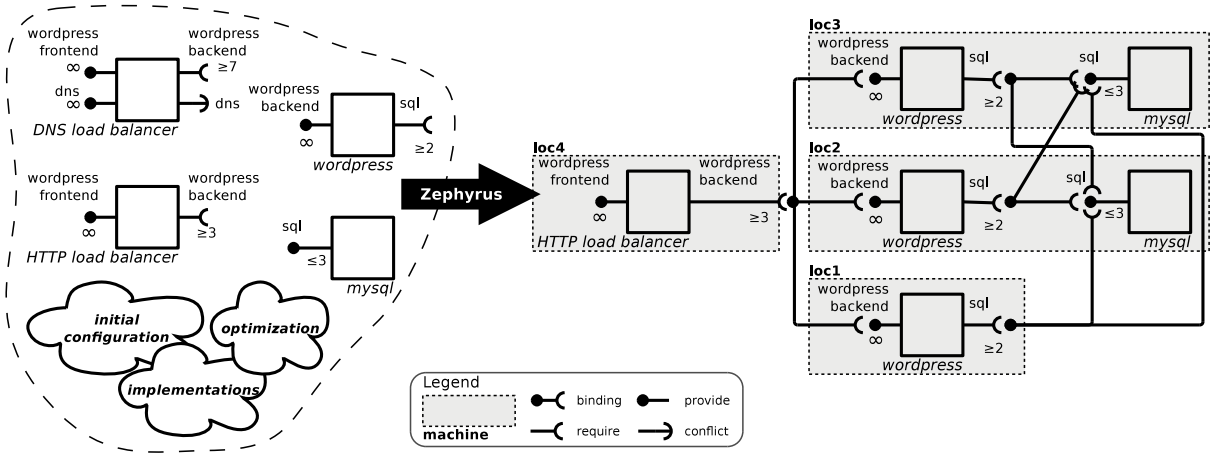

Figure 1: `Zephyrus` used to design a scalable, fault-tolerant Wordpress deployment

- no more than 1 DNS server deployed in the administrative domain;
- different Wordpress (and MySQL) instances are deployed at different locations.[1]

Similar constraints might exist on machine resources, e.g. we expect Varnish to consume 2Gb of RAM and we don't want to deploy it to a smaller machine, especially if in combination with other RAM-consuming services. Note that "house" requirements are not intrinsically related to the software components we are using, but are rather an encoding of explicit architectural choices.

2.2 Architecture synthesis

Figure 1 shows the application of our toolchain to the design of a complex Wordpress deployment like the one we have discussed.

On the left of the black arrow is a schematic representation of `Zephyrus`' input, on the right its output. Available services are depicted in the figure using a graphical syntax inspired by the Aeolus model [9], each one with its own requirements, conflicts, and (house) replication policy. Component requirements are exposed as required ports that should be connected, via bindings, to matching provided ports offered by other service instances, respecting port replication constraints: an upper bound (or ∞) on the amount of incoming bindings for provided ports; a lower bound on the amount of outgoing bindings to *different* service instances for required ports. For example, the fact that the HTTP load balancer requires 3 Wordpress replicas is indicated by the ≥ 3 annotation on its `wordpress backend` required port, and the fact that the DNS load balancer is incompatible with other DNS services is indicated by the `dns` conflict port.

Note that our ports result in a very flexible notion of dependency, with *choice*: any requirement can be satisfied by *any* component providing the right port. For instance, if we require the port `wordpress frontend`, we allow `Zephyrus` to choose which of `DNS load balancer` or `HTTP load balancer` is the best to use. To our knowledge, `Zephyrus` is the only tool to manage such flexibility in dependencies.

[1]It is technically possible to co-locate multiple, say, MySQL instances on the same machine, but it would be pointless to do so when we are seeking fault tolerance and load balancing.

Services and implementations. `Zephyrus` takes as input a description of the available service types, and an *implementation* relation that maps each *service* to the set of packages implementing it.[2] These two parts of the universe are given as input to `Zephyrus` as a JSON file that in our running example looks like this:

```
{ "component_types": [
  { "name"   : "DNS-load-balancer",
    "provide" : [["@wordpress-frontend"], ["@dns"]],
    "require" : [["@wordpress-backend", 7]],
    "conflict": ["@dns"],
    "consume" : [["ram", 128]] },
  { "name"   : "HTTP-load-balancer",
    "provide" : [["@wordpress-frontend"]],
    "require" : [["@wordpress-backend", 3]],
    "consume" : [["ram", 2048]] },
  { "name"   : "Wordpress",
    "provide" : [["@wordpress-backend"]],
    "require" : [["@sql", 2]],
    "consume" : [["ram", 512]] },
  { "name"   : "MySQL",
    "provide" : [["@sql", 3]],
    "consume" : [["ram", 512]] } ],
  "implementation": [
    [ "DNS-load-balancer", ["bind9"] ],
    [ "HTTP-load-balancer", ["varnish"] ],
    [ "Wordpress", ["wordpress"] ],
    [ "MySQL", ["mysql-server"] ] ] }
```

The `component_types` section describes the available component types with their ports, as well as their non functional requirements like memory or bandwidth. Port names are distinguished from components or packages by a simple syntactic convention: ports start with `@`. The `implementation` section maps services to the software packages that should be installed to realize them on actual machines.

Package repositories. Unlike other tools, `Zephyrus` is fully aware of available package repositories, with their dependencies and conflicts, and uses such information to ensure that package-level conflicts and dependencies are respected on all machines. It is possible to associate different package repos-

[2]In the example we have kept things simple, but `Zephyrus` is capable of handling complex situations where the same service can be implemented by different packages on different machines, according to the locally installed OS.

Table 1 Specification Syntax

S	::=	`true` | e *op* e	Specification
	|	S `and` S | S `or` S	
	|	S `=>` S | `not` S	
e	::=	n | $\#\ell$ | $n \times e$	Expression
	|	$e + e$ | $e - e$	
ℓ	::=	k | t | p	Elements
	|	$(J_\phi)\{J_r : S_l\}$	
S_l	::=	`true` | e_l *op* e_l	Local Specification
	|	S_l `and` S_l | S_l `or` S_l	
	|	S_l `=>` S_l | `not` S_l	
e_l	::=	n | $\#\ell_l$ | $n \times e_l$	Local Expression
	|	$e_l + e_l$ | $e_l - e_l$	
ℓ_l	::=	k | t | p	Local Elements
J_ϕ	::=	_ | o *op* $n; J_\phi$	Resource Constraint
J_r	::=	_ | r | $r \vee J_r$	Repository Constraint
op	::=	$\leq$ | $=$ | $\geq$	Operators

itories to different locations, allowing to handle deployment of heterogeneous systems. The size of a repository may be huge (the Debian Squeeze repository contains ≈30'000 packages), so `Zephyrus` uses `coinst` [8] to abstract packages into a set of much smaller equivalence classes, and yet sufficient to capture all package incompatibilities.

Available (virtual) machines. Another essential part of `Zephyrus` input is the description of the initial configuration, i.e. the set of available machines with information on their resources: memory, package repository, existing services and packages. In our example, we start with an initial configuration consisting of 6 bare locations with 2Gb of RAM. Such configuration is fed to `Zephyrus` in JSON format, e.g.:

```
{ "locations" : [
    { "name" : "loc1",
      "repository" : "debian-squeeze",
      "provide_resources" : [["ram", 2048]] },
    { "name" : "loc2",
      "repository" : "debian-squeeze",
      "provide_resources" : [["ram", 2048]] },
    [...] }
```

Target system specification. `Zephyrus` accepts a specification of the desired target system. Specifications are defined according the abstract syntax presented in Table 1.

A specification S is a set of basic constraints e *op* e, combined using the usual logical connectors. These basic constraints specify how many elements (packages, component types, etc) should be in the generated configuration, using terms of the form $\#\ell$ that correspond to the number of instances of element ℓ in the system. For instance, one might state that we want at least 3 instances of the component type `apache`: "`#apache` $\geq$ `3`", where `#apache` represents the number of `apache` instances in the configuration.

Moreover, it is possible to express constraints on locations. Locations can be specified in our syntax with the term $(J_\phi)\{J_r : S_l\}$ where J_ϕ is the constraint on the resource available on that machine; J_r is the set of repositories that can be installed on that machine ('_' standing for any repository); and S_l is a constraint specifying the contents of the machine (basically, S_l is S without locations). For instance, we can specify that no location with less than 2Gb of RAM and `redhat` installed should have a `MySQL` running: "`#(mem < 2G){redhat: #MySQL` $\geq$ `1} = 0`".

For our running example we need exactly one Wordpress frontend (i.e., exactly one service offering a `wordpress-frontend` port), and that no machine is deployed with more than one instance of either MySQL/Wordpress services on it.

```
(#@wordpress-frontend = 1)
and #(_){_ : #MySQL > 1} = 0
and #(_){_ : #Wordpress > 1} = 0
```

Note that no constraint is imposed on the co-location of *different* services on the same machine.

Optimization criteria. In `Zephyrus`, one may request a solution that is optimal w.r.t. a specific objective function. Currently, `Zephyrus` supports two built-in optimization criteria, namely `compact` and `conservative`, which respectively minimize the number of components and locations used, or their *difference* with respect to the initial configuration.

Running Zephyrus. We are now ready to ask `Zephyrus` to compute the final configuration:

```
$ zephyrus -u univ.json -opt compact \
   -ic conf.json -spec sp.spec \
   -repo debian-squeeze ds.coinst
```

In addition to the obvious parameters (universe, optimization function, configuration, specification), we pass an extra one: the `-repo` option tells `Zephyrus` that all the information about the packages contained in the repository named `debian-squeeze` is available in the file `ds.coinst`.

The actual output of `Zephyrus` contains a complete description of the system to be deployed; it is too long to be listed here in full, so we only highlight some excerpts of it. The format is the same as for configurations, and starts with the description of the locations:

```
{ "locations": [
    { "name": "loc1",
      "provide_resources": [ [ "ram", 2048 ] ],
      "repository": "debian-squeeze",
      "packages_installed": [ "wordpress" ] },
    { "name": "loc2",
      "provide_resources": [ [ "ram", 2048 ] ],
      "repository": "debian-squeeze",
      "packages_installed": ["mysql-server",
        "wordpress" ] }                          [...]
```

We see that each location is associated to a list of packages that should be installed there. Only the root packages are listed, and `Zephyrus` has already checked that they can be co-installed, satisfying dependencies and conflicts.

The second part of the output is the list of service instances present in the system, mapped to their locations:

```
  "components": [
    { "name": "Wordpress-1",
      "type": "Wordpress",
      "location": "loc1" },
    { "name": "Wordpress-2",
      "type": "Wordpress",
      "location": "loc2" },
    { "name": "MySQL-1", "type": "MySQL",
      "location": "loc2" },                [...]
```

Finally, the third part of the output lists the bindings that connect (ports of) service instances together:

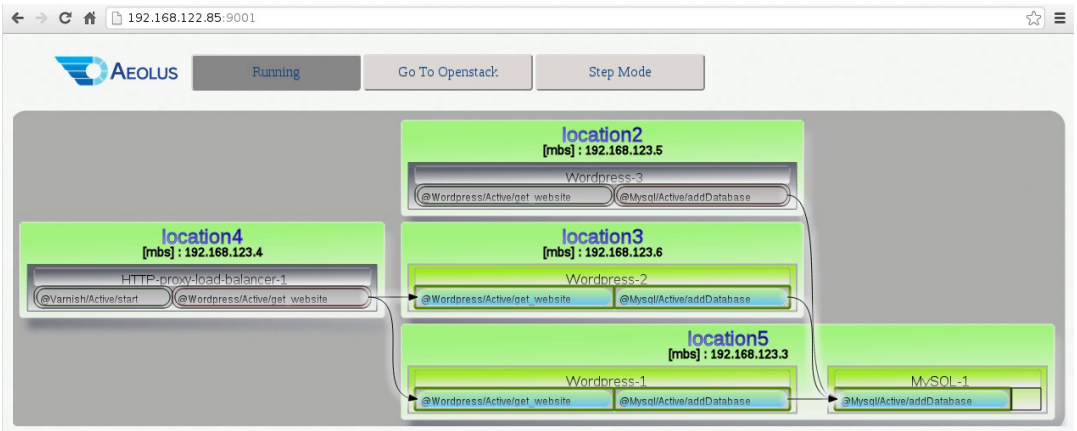

Figure 2: Screenshot of `Armonic` web interface.

```
"bindings": [
  { "port": "@wordpress-backend",
    "requirer": "HTTP-load-balancer-1",
    "provider": "Wordpress-1" },
  { "port": "@wordpress-backend",
    "requirer": "HTTP-load-balancer-1",
    "provider": "Wordpress-2" },
  { "port": "@sql",
    "requirer": "Wordpress-1",
    "provider": "MySQL-1" }            [...]
```

The configuration corresponding to `Zephyrus` output is depicted on the right of Figure 1, where shaded boxes denote locations; we omit installed packages for the sake of readability. All choices there—load balancer, mapping between services and machines, bindings, etc.—have been made by `Zephyrus`. Note how services have been co-located where possible, minimizing the number of used machines: only 4 out of the 6 available machines have been used: the proposed solution is optimal w.r.t. the desired metric.

2.3 Configuration deployment

Once we have the configuration generated by `Zephyrus` as output, we are left with the task of turning it into a running system. While `Zephyrus` output is agnostic as to the final deployment tool, we have developed our own tool—called `Armonic`—to work closely in conjunction with `Zephyrus`.

Starting from the configuration generated by `Zephyrus`, `Armonic` first provisions the needed virtual machines (VMs) on a target private OpenStack cloud, creating new instances as needed. The resource information (CPU, storage, memory) contained in `Zephyrus` output are used by `Armonic` to choose appropriately-sized VMs (AKA "flavors").

After provisioning, `Armonic` takes care of *configuring VMs* using an agent-orchestrator architecture: each VM comes with an agent which receives configuration instructions from a central orchestrator. During VM configuration, `Armonic` installs on each VMs the packages dictated by `Zephyrus`; tunes service configuration files to implement bindings (e.g. to "connect" a Wordpress to a MySQL, `Armonic` will patch a Wordpress configuration file to point to a given VM IP address and port); and starts services in the right order as it goes. In our example, a Wordpress instance is deployed at location `loc1`. However, a MySQL database must be available *before* the deployment of Wordpress in order to properly start the service. To address this, `Armonic` devises a *deployment plan*, using bindings to determine a suitable deployment ordering, and follows it during component deployment. In the example `Armonic` will deploy Mysql right away (as it has no further component dependencies), then Wordpress, and finally the load balancer.

The `Armonic` orchestrator is equipped with a web interface, shown in Figure 2 at work on the deployment of a simplified version (using a single MySQL database, instead of a cluster) of the configuration of Figure 1. To deploy an application, users can simply feed `Zephyrus` output into the `Armonic` web interface and then monitor live the state of deployment. In Figure 2 the deployment is almost finished: all components are deployed and connected, except the last Wordpress instance and the load balancer which are shown grayed-out, as they are only partially deployed. The deployment of this use case takes about 7 minutes, including building and booting virtual machines, package installation, and service configuration.

3. TOOLCHAIN INTERNALS

3.1 Minimizing input

Figure 3 presents a simple schema of the architecture of `Zephyrus`, which is basically structured into five blocks. The input phase of `Zephyrus` collects all the data provided by the user.

For each location, we take into account not only the available services, but also all the possible ways to deploy them (i.e. packages that must be installed to realize them, together with their dependencies): this can amount to handle tens of thousands of packages for each location, and a naïve approach would simply be unfeasible. We believe this is one of the fundamental reasons why competing tools do not represent package relationships explicitly or completely, with the consequence of potentially producing configurations which are not deployable due to package incompatibilities unknown to the tool. We compare this aspect of `Zephyrus` with alternative approaches in Section 5.

To render the problem tractable, `Zephyrus` performs several simplification passes on the input data that greatly reduce its size: the universe is trimmed by removing all services that are not in the transitive closure of the services present in the initial configuration or the request; package

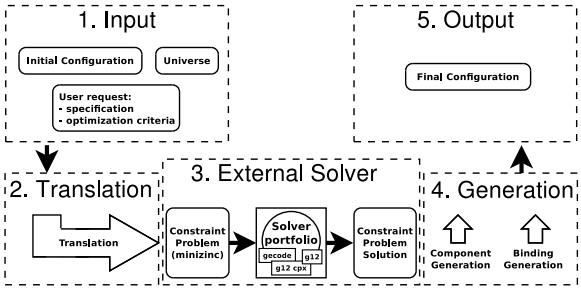

Figure 3: Overall architecture of `Zephyrus`

repositories are pruned by keeping only packages that implement services which were not removed by the previous simplification phase; lower and upper bounds on the needed resources and components are computed, and only the minimum estimated number of available locations is kept. All these operations are safe, as we can prove that they do not exclude any correct solution.

A second important simplification is achieved by using a slightly modified version of the `coinst` tool [8], which reduces by several orders of magnitude the data present in software package repositories, like those offered by the Debian or RedHat distributions, while retaining all the *coinstallability* information needed to determine if a set of packages can or cannot installed together. We refer the interested reader to [8] for precise figures and detailed proofs; we just recall here that this simplification is safe, and preserves all correct solutions.

3.2 Constraint generation

The second phase of `Zephyrus` translates the (trimmed) input into a set of constraints over non-negative integers. These constraints use different variables for the number of instances to create on each of the locations for each of the types in the universe, and also variables representing the packages that must be installed on each location. The constraints impose that the instances respect the definition of their type in the universe, the way how instances are implemented by packages, the (compacted) dependencies and conflicts between packages, and the problem specification.

The most interesting of these constraints ensure that it is possible to create the bindings between all instances according to the capacity constraints. These constraints distinguish our approach from others [13, 12, 14], they are necessary due to the flexible dependencies we have on ports. Constraints are constructed using auxiliary variables $B(p, t_r, t_p)$ for the number of bindings on port p between requesting instances of type t_r and providing instances of type t_p.

On the example of Section 2.2, these particular constraints for the bindings on port `sql` look like this:

$$B(\texttt{sql}, \texttt{wp}, \texttt{mysql}) \leq \texttt{\#mysql} * 3 \quad (1)$$

$$B(\texttt{sql}, \texttt{wp}, \texttt{mysql}) \geq \texttt{\#wp} * 2 \quad (2)$$

$$B(\texttt{sql}, \texttt{wp}, \texttt{mysql}) \leq \texttt{\#wp} * \texttt{\#mysql} \quad (3)$$

Here, (1) expresses that the number of bindings on port `sql` between instances of the two types is at most the number of instances of the providing type `mysql` times 3 (since any component of that type can bind to at most 3 instances), (2) that the number of bindings on port `sql` is at least the number of instances of the requesting type `wp` times 2 (the number of binding each component of type `wp` requires), and finally (3) states that the number of bindings is at most the number of pairs of instances of type `wp` with instances of type `sql`. This last restriction expresses that no two bindings may exist between the same pair of instances.

The ability to capture as simple integer constraints the existence of a complete architecture corresponding to a specification is the cornerstone of our approach. It allows us to deal with the many facets of a system design as a whole, and thus ensures the completeness of our tool and the optimality of the generated configuration. In particular, if the output of `Zephyrus` indicates that several services are to be installed on the same machine, we know that no conflict between the packages that realize them will arise on actual machines.

3.3 External solvers

The generated constraints, as well as the optimization function, are expressed using the `MiniZinc` constraint modelling language [24, 22]. This allows us to employ any of the many existing constraint solvers that support `MiniZinc`. `Zephyrus` can currently use GeCode [29] (an efficient open source solver) or several of the solvers provided in the G12 suite [30]. The tool exploits this flexibility by implementing a *solver portfolio* approach [2, 3] that reduces execution time by running several solvers in parallel, and stops as soon as one of the solvers finds a solution.

3.4 Configuration generation

When the external solver finds a solution for the generated constraint, the next part of `Zephyrus` proceeds to transform that solution, which is a simple mapping from variables to integers, into an actual configuration. The two main challenges of that generation are: i) to reuse as many existing parts of the initial configuration as possible in order to minimize the impact on the existing system; and ii) to correctly generate bindings between the instances taking into account that any two instances can be bound on a given port at most once. The algorithms employed in this generation phase are presented in detail in [7].

Once the configuration has been generated, it can be written to a file in two different formats: either (a) the same *JSON* format used for the input configuration, which precisely describes all the configuration features and is used by `Armonic` as input; or (b) the *dot* format that encodes the configuration into a graph that can be viewed using the `dot` program to visualize the synthesized architecture.

If no configuration satisfies the given input constraints, `Zephyrus` will exit with an error message and produce no output files.

3.5 Synthesis soundness and completeness

An important property of `Zephyrus` is that all its parts have been formalized. In particular, the translation into constraints and the generation of the configuration, two very complex and important pieces of `Zephyrus`, have been precisely described and proven correct in [7], where the following results have been shown for `Zephyrus`:

Theorem 1 (Soundness)**.** *The configuration generated by `Zephyrus` is correct w.r.t. the input universe and specification.*

Theorem 2 (Completeness)**.** *If there exists a configuration that validates the input universe and specification, then `Zephyrus` will successfully generate a correct configuration.*

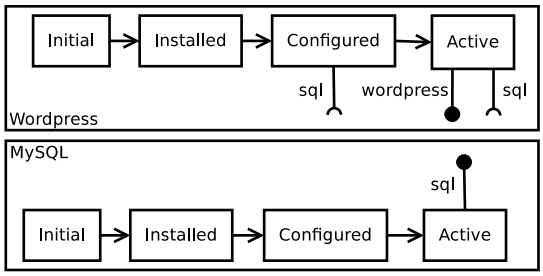

Figure 4: `Armonic` representation of Mysql and Wordpress component types.

Theorem 3 (Optimality)**.** *The configuration generated by `Zephyrus` is optimal w.r.t. the input optimization function.*

3.6 Deployment planning

Virtual machine selection. In this paper, `Armonic` uses the popular OpenStack platform to provision virtual machines, starting from the *locations* entries found in `Zephyrus` output to determine machine names and resources (compute, storage, and memory capacity). In order to map resources to the available VM "flavors", a correspondence table is defined between `provide_resources` and Openstack flavors, for instance:

```
[["ram", 2048]] -> m1.small
```

Using this table, `Armonic` can create VMs using the Openstack API, e.g.:

```
nova boot --flavor m1.small --image debian-squeeze loc1
```

Component life-cycle. `Armonic` associates each component type to an acyclic state diagram that captures the component life-cycle. As an example, Figure 4 shows the life-cycles for the Worpdress and Mysql component types that we have used in the walkthrough of Section 2.

Different states may require and provide different ports. For instance, MySQL's active state provides a port `@sql`, denoting that such port can be depended upon only when that state in the MySQL life-cycle has been reached. Given that Wordpress' configured (and subsequent) state(s) require that port, Wordpress cannot enter such state before MySQL has entered its active state.

Determining deployment order. Section 2.3 briefly introduces the need of a component deployment plan. Computing such a plan can be quite challenging (even undecidable [9]), depending on the expressivity of component constraints. Hence, `Armonic` makes several simplifying assumptions to keep the problem tractable.

First, `Armonic` currently assumes that all VMs are "empty", with no services initially deployed on them. This assumption simplifies deployment of services because it ignores reconfiguration needs. Second, we suppose there is only one path to reach a given state, while `Armonic` life-cycle representation allows for multiple paths. Finally, we suppose that there are no circular dependencies between components. We are working on an improved planning algorithm which will address these limitations [18] and also allow for optimizations such as maximally parallel component deployment.

3.7 Service configuration

To connect components according to `Zephyrus` bindings, `Armonic` has to generate service configuration files and may have to create resources. For instance, connecting MySQL to Wordpress consists of creating a MySQL database and user with the appropriate permissions, and making a Wordpress configuration file point to the right IP address, port, DB name, and user name.

To automate this process, `Armonic` allows to attach additional information to provide and require ports. Thus, the provide port `@sql` of MySQL exposes three required variables, a database name, a user name, and a user password. The require port `@sql` of Wordpress exposes these variables with predefined values. Since the component MySQL is the provider of the bindings `@sql` which has Wordpress as requirer, `Armonic` uses this information to bind requirer and provider configuration variables. In this case, `Armonic` will call the provide port `@sql` of MySQL with values of require port `@sql` of Wordpress. This action will create the database and the MySQL user. These values will then be used by the `Armonic` agent to patch the Wordpress configuration file.

4. EXPERIMENTATION

The complete toolchain presented in this paper is available as free software, released under the GPL license. `Zephyrus` amounts to about 10.000 lines of OCaml [20] and is available at `https://github.com/aeolus-project/zephyrus/`; `Armonic` is about 5.000 lines of Python, plus glue code for component life-cycles written in shell script or Augeas [28], and is available at `https://github.com/armonic/armonic`. We have experimented the complete toolchain in both artificial and industrial settings; in this section we present some of our findings. As the figures for `Armonic` are dominated by the deployment time used by system-level tools (package managers, service startup, etc.), and also because we have already briefly presented them at the end of Section 2.3, we focus here on `Zephyrus`.

4.1 Synthesis efficiency

Given that the architecture synthesis part of our toolchain implemented by `Zephyrus` has a daunting complexity in theory one may ask whether this part could be a bottleneck of our approach. In order to answer this question we have conducted several architecture synthesis benchmarks on both realistic and extreme use cases. The ones we illustrate here are variants of the WordPress example described in Section 2. There are, however, three important changes needed to properly benchmark it:

- The use case is parameterized to scale it up and to demonstrate how `Zephyrus` handles problems which require more and more components and locations: (i) the first parameter is the minimum replication constraint on the `wordpress backend` port (required by the load balancer); (ii) the second parameter is the minimum replication constraint on the `sql` port (required by WordPress components).

- The resources associated to available locations are inspired by Amazon's EC2 VM offering. We took what

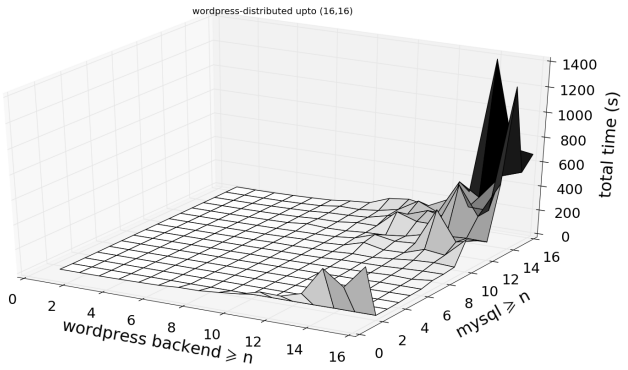

Figure 5: WordPress synthesis benchmarks, for increasing values of the replication constraints on wordpress backends (x-axis) and mysql (y-axis)

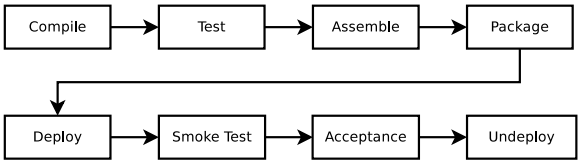

Figure 6: local qualification process at Kyriba

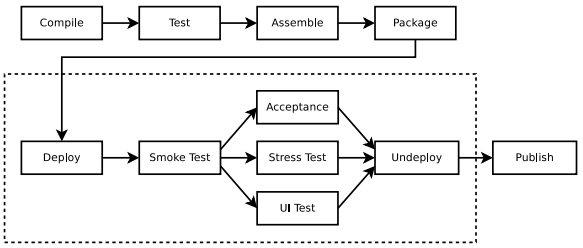

Figure 7: remote qualification process at Kyriba

Amazon calls "old" (previous generation) general purpose machines. So there are four types of locations available, corresponding (by cost and capacity) to Amazon instance types: `m1.small`, `m1.medium`, `m1.large` and `m1.xlarge`. We have provided `Zephyrus` with a finite, but large enough number of machines (250 for each instance type).

- We use a single package repository associated to each machine, and a single package implementing all the available component types. This simplification does not affect the test results, as all component types in this benchmark are co-installable anyhow.

We have used a portfolio of solvers, as discussed in Section 3.3, consisting of the following 3 solvers: Gecode [29], the standard finite domain constraint solver from the G12 suite [30], and the G12/CPX (Constraint Programming with eXplanations) solver from the G12 suite. As these solvers are optimized for different goals, each of them works better in some situations and worse in others. It is very difficult to guess beforehand which solver is more adapted to a specific constraint problem instance. The portfolio approach permits us to work around this obstacle by trying these three approaches at the same time.

We have varied the two use case parameters from 1 to 16. Execution times are obtained as average of 5 runs on a commodity desktop machine (Intel i7 3.40 GHz, 8 GB of RAM). The diagram in Figure 5 shows that a vast majority of cases are solved very quickly in less than one minute. Only the larger ones can take more than 20 minutes, e.g. the `(14,16)` case, which is the highest peak in the chart. To put this worst-case solving time into perspective, please note that the largest use case (`(16,16)`) consists of 103 components, interconnected by 272 bindings, and distributed over 86 machines. This surpasses by a significant margin the size of most professional WordPress deployments.

4.2 Application to continuous integration

`Zephyrus` has been deployed in a large industrial use case at Kyriba Corporation[3], a large software editor providing Software-as-a-Service treasury management solutions. In the following we offer a return on that experience, validating the usefulness of the proposed approach in an industrial setting. This use case highlights the importance of considering all system design constraints together and the benefits of statically detecting when they are not satisfiable—in which case `Zephyrus` will exit with an error. It also shows the flexibility of our toolchain, by relying on a (in-house) deployment back-end other than `Armonic` for the actual deployment.

Kyriba solution is a complex software platform composed of more than 150 components deployed on multi-tier architectures, with many different versions running at the same time. Maintaining the consistency of the system as a whole is a major undertaking. To address this challenge, Kyriba has invested in completely automating the build, integration, and deployment processes.

Kyriba distinguishes two software qualification processes: a local one run by individual developers on their machines; and another, more thorough one run on a remote continuous integration (CI) service. Heavy, exhaustive tests are performed remotely, whereas individual developers only run a subset of available tests on their machines.

Successful completion of the local qualification process, detailed in Figure 6, is required in order to be able to commit code changes to the source version control system. After each commit the process depicted in Figure 7 is triggered: first CI runs the same process that has been run on developers machine; automatic deployment is then performed on the cloud infrastructure with the latest component version, and more extensive tests—UI, deep functional scenarios, stress tests—are executed.

4.2.1 A Case for automation

Kyriba follows the continuous integration recommendations [10] and implements acceptance and stress tests. These tests are very time consuming: while local tests take less than 4 minutes to complete, global ones might take 4–8 hours. Furthermore, as *Kyriba solution* is an assembly of multiple components, integration tests involve many interdependent components that should all be deployed before testing. When deploying on a single machine, maintaining

[3] `http://www.kyriba.com/`

consistency (e.g. version alignment) is rather easy and can be enforced using package dependencies; when components are distributed as services on multiple physical/virtual machines, consistency is harder to maintain.

In the past, test deployment was done using custom tools involving a manual setup, and component/protocol incompatibilities were only detected at runtime. Short feedback loops discipline helps developers with error diagnostic related to small code changes [16], so Kyriba has been looking for a tool that could anticipate error detection.

`Zephyrus` turned out to be a perfect fit for this need, as a deployment validation tool for both the local and remote qualification processes. `Zephyrus` is now used in distributed component consistency validation and deployment configuration scenarios. `Zephyrus` helps to get feedback before launching local deployments tests and has led to a significant reduction of the number of failures occurring during automated deployment in comparison to the previous, more manual, test setup.

4.2.2 Zephyrus adoption

For developers. Developers define relationships between components in partial `Zephyrus` files when they create packages for their project. These files are then merged with `Zephyrus` files containing the full infrastucture description provided by engineering operations team before being processed by the solver.

Two kinds of such relationships need to be defined:

- Dependencies between packages with specific version requirements, e.g. the application 1.0 requires a web server of version at least 3.2.4 to be installed on the same machine to run properly;

- Service binding relationship with API level requirement, e.g. the application 1.0 requires a service API version 1 exposed by another application on some machine, not necessarily where the application is deployed.

Developers can declare these requirements using ports specified in the `Zephyrus` universe definition:

```
{ "component_types": [
  { "name"  : "fa-accounting-engine-0.1",
    "provide": [["@fa-accounting-engine-v1", ["
     FiniteProvide", 1]]],
    "require": [["@graphite-v3", 1]] } ],
  "implementation": [
    [ "fa-accounting-engine-0.1",
      [["debian-kyriba",
        "kyriba-fa-accounting-engine (= 0.1)"]] ] ] }
```

The component type `kyriba-fa-accounting-engine` provides a "fa-accounting-engine-v1" service with API level 1 and requires a "graphite-v3" service. In the implementation section, the `component_type` is linked to the concrete package implementation `kyriba-fa-accounting-engine (= 0.1)`. This partial `universe` definition is merged with the full universe description (containing the definitions of all Kyriba components) and the default specification in order to verify that at least one final configuration exists. This ensures that no dependency problem can arise during deployment.

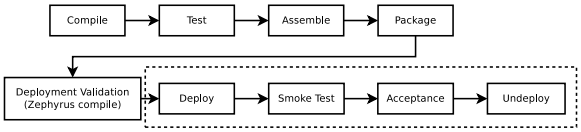

Figure 8: local qualification process with Zephyrus

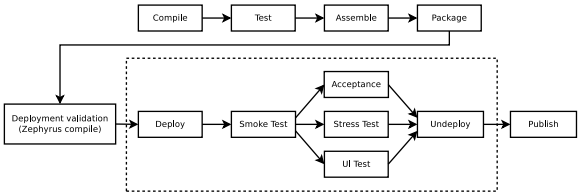

Figure 9: global qualification process with Zephyrus

Developers may also override the default `specification` with their own specification to validate different deployment scenarios during the local qualification process.

The local qualification process is modified by adding a `Zephyrus` validation stage before (local) deployment, see Figure 8. Using `Zephyrus` metadata developers simply declare component interfaces and the way they are exposed. Moreover, using `Zephyrus`, developers know beforehand if the components they are working on can be deployed together with other components.

For continuous integration and deployment environment. Similarly to the local one, the global qualification process has been instrumented with an extra `Zephyrus` validation stage as illustrated in Figure 9.

`Zephyrus` automatically checks application consistency before actually engaging in a deployment process. If the solver finds a solution, the related configuration is used by infrastructure management scripts based on the Fabric library[4] in order to orchestrate an integration test deployment on the cloud infrastructure (such as the Amazon Elastic Cloud Computing service). The newly created platform is then checked against all acceptance, stress and UI test cases.

To upgrade the production platform. `Zephyrus` is also used to plan platform upgrades on the infrastructure currently in production. According to the software road map, product managers define the component versions needed to be shipped to production for a milestone release. All those values are set in `Zephyrus` files, and based on the current production deployment, `Zephyrus` computes an output file containing the different application packages that should be installed with the related configuration parameters that need to be set for application binding. This guideline file is then used by engineering teams to write orchestration scripts and pinpoint manual upgrade tasks.

4.2.3 Outcome

Summing up, Kyriba's experience with `Zephyrus` is that, instead of managing deployment scenarios manually using spreadsheets and flat documents, with *ad hoc* semantics leading to complex, time-consuming and error-prone deployments, `Zephyrus` brings precise semantics and simplifies the

[4] `http://www.fabfile.org`

automation of software qualification processes. `Zephyrus` provides static validation before actually performing expensive and very long dynamic validation at runtime. As most "compiler-like" tools, `Zephyrus` improves engineering quality and reduces building cost with less failures at deployment, integration test stage and platform upgrade.

5. RELATED WORK

The problem of managing networks of interconnected machines has attracted significant attention in the area of system administration. Many popular system management tools exist to that end: CFEngine [4], Puppet [17], MCollective [27], and Chef [26] are just a few among the most popular ones. Despite their differences, such tools allow to declare the components that should be installed on each machine, together with their configuration files. Then, they employ various mechanisms to deploy components accordingly. The burden of specifying where components should be deployed, and how to interconnect them is left to the sysadmin, let alone the difficult problem of optimal resource allocation. As an extra complication, system management tools stop at the package management abstraction, and therefore have no way of knowing *in advance* whether deployment will succeed or not. If the sysadmin requests to install two incompatible web servers on the same machine, the incompatibility will only be discovered by the package manager at deploy time, when one of the two services fails to get installed (or started). At that point, it is up to the admin to go back to the planning stage and work around the incompatibility. In our approach all package relationships are known to `Zephyrus` which can therefore plan around component incompatibilities.

System management tools can be used as an alternative to `Armonic`, though. Once optimal resource allocation is done by `Zephyrus`, the actual deployment can be delegated to them, now with the guarantee that no deployment error due to incompatibilities will arise; an interesting candidate could be [31].

CloudFoundry [32] specifically targets application deployment in the "cloud", but suffers from the same limitations as described above. ConfSolve [15] improves on the system management approach: it relies on a constraint solver to propose an optimal allocation of virtual machines to servers, and of applications to virtual machines, but it does not handle connections among services, nor capacity or replication constraints, and is unaware of package incompatibilities.

In Juju [5], each service is deployed on a single machine (or, more recently, in a virtual container on a machine). That avoids the issue of component incompatibilities, but does so at the price of wasting resources. In our Wordpress example `Zephyrus` proposes a solution that needs 4 machines, whereas Juju would have required 6.

Two recent efforts, Feinerer's work on UML [12] and Engage [14], are more similar to our approach as they both rely on a solver to plan deployments. Feinerer's work is based on the UML component model, which includes conflicts and dependencies with capacity constraints, but uses dependencies only between components, which greatly restricts the expressiveness of the model (choices are not possible). Moreover, no tool for actually building the computed configuration is provided. Engage, on the other hand, offers no support for conflicts in the specification language: one can only indicate that a service can be realized by exactly one out of a list of components. Neither Feinerer's work nor Engage, or any other tool that we are aware of, allows to find a deployment that uses resources in an optimal way, minimizing the number of needed (virtual) machines.

Another approach to automated deployment is proposed in [11], which uses an Architecture Description Language with user-provided information about relationships among software services, and implements a decentralized protocol to perform automatic configuration. This work may also be used as a backend for `Zephyrus`.

Finally, we would like to put `Zephyrus` in perspective. In our view, automated management of cloud applications is best realized by a 2-phase approach. In the first phase (*architecture synthesis*) `Zephyrus` or similar tools are used to devise an optimal system architecture. Then, in a second phase (*planning*), the obtained configuration is compared with that of the existing system to produce a detailed deployment plan that migrates the existing system to the desired one. To implement planning, the actual state of services and their life cycles (e.g. how do they pass from an inert "installed" state to an "up and running" one? do dependencies and conflicts change in the meantime?), ignored for the purpose of this paper, become relevant again. Even though planning has been shown to be undecidable in the most general case [18], promising progress has been made on automated planning for restricted cases, like planning in the presence of complex activation requirements that include circular dependencies, whereas giving up the possibility of expressing component conflicts [19].

6. CONCLUSION

We have introduced an automated approach to the design and deployment of complex distributed applications composed of interconnected services, as typically found in modern "cloud" environments. The system architect can specify the components needed to obtain the required functionalities, add non-functional constraints—e.g. maximum number of client components connected to a given service, or minimum number of replicas—as well as available physical resources—e.g. memory or bandwidth—and declare component incompatibilities. The architect can also choose an optimization goal, allowing to specify whether she prefers a conservative solution that changes the current configuration as little as possible, or a minimum-cost solution.

The approach is realized by two complementary tools: `Zephyrus` will synthesize an optimal architecture, including precise information about service interconnections. Such an architecture is then fed to the second tool, `Armonic`, which is able to deploy it on state-of-the art cloud infrastructures such as OpenStack, taking care of all deployment aspects from machine provisioning to service configuration and initialization. A major advantage of the proposed approach w.r.t. the state of the art is that all existing constraints, including software package-level incompatibilities, are taken into account to prevent deploy-time errors. We have validated the complete toolchain both theoretically, showing soundness and completeness of the approach, and practically by applying it to relevant industrial use cases.

To the best of our knowledge this toolchain is the first that allows to consistently handle capacity and replication constraints, conflicts, and service co-location, thus finally providing an instrument able to handle the stringent requirements of cloud applications in the real world.

7. REFERENCES

[1] A. Aleti, B. Buhnova, L. Grunske, A. Koziolek, and I. Meedeniya. Software architecture optimization methods: A systematic literature review. *IEEE TSE*, 39(5):658–683, 2013.

[2] R. Amadini. Evaluation and application of portfolio approaches in constraint programming. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5-Online-Supplement), 2013.

[3] R. Amadini, M. Gabbrielli, and J. Mauro. An empirical evaluation of portfolios approaches for solving CSPs. In C. P. Gomes and M. Sellmann, editors, *Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 7874 of *LNCS*, pages 316–324, 2013.

[4] M. Burgess. A site configuration engine. *Computing Systems*, 8(2):309–337, 1995.

[5] Canonical Ltd. Juju, devops distilled. `https://juju.ubuntu.com/`. Retrieved October 2013.

[6] M. Catan, R. Di Cosmo, A. Eiche, T. A. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Aeolus: Mastering the complexity of cloud application deployment. In *ESOCC 2013: Service-Oriented and Cloud Computing*, volume 8135 of *LNCS*, pages 1–3, 2013.

[7] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, and J. Zwolakowski. Optimal provisioning in the cloud. Technical report, Université Paris Diderot, 2013. Available at `http://hal.archives-ouvertes.fr/hal-00831455/`.

[8] R. Di Cosmo and J. Vouillon. On software component co-installability. In *Foundations of Software Engineering (FSE)*, pages 256–266. ACM, 2011.

[9] R. Di Cosmo, S. Zacchiroli, and G. Zavattaro. Towards a formal component model for the cloud. In *Software Engineering and Formal Methods (SEFM) 2012*, volume 7504 of *LNCS*, pages 156–171, 2012.

[10] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[11] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-configuration of distributed applications in the cloud. In *International Conference on Cloud Computing*, pages 668–675. IEEE, 2011.

[12] I. Feinerer. Efficient large-scale configuration via integer linear programming. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing (AI EDAM)*, 27(1):37–49, 2013.

[13] I. Feinerer and G. Salzer. Consistency and minimality of UML class specifications with multiplicities and uniqueness constraints. In *Theoretical Aspects of Software Engineering (TASE)*, pages 411–420, 2007.

[14] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: a deployment management system. In *PLDI'12: Programming Language Design and Implementation*, pages 263–274. ACM, 2012.

[15] J. A. Hewson, P. Anderson, and A. D. Gordon. A declarative approach to automated configuration. In *LISA '12: Large Installation System Administration Conference*, pages 51–66, 2012.

[16] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[17] L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1):19–25, 2006.

[18] T. A. Lascu, J. Mauro, and G. Zavattaro. Automatic component deployment in the presence of circular dependencies. In *Formal Aspects of Component Software (FACS) 2013*, volume 8348 of *LNCS*, 2013.

[19] T. A. Lascu, J. Mauro, and G. Zavattaro. A planning tool supporting the deployment of cloud applications. In *International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 213–220. IEEE, 2013.

[20] X. Leroy, D. Doligez, J. Garrigue, and D. Rémy. *The Objective Caml system release 4.01; Documentation and user's manual*. INRIA, Rocquencourt, Paris, 2013.

[21] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *International Conference on Automated Software Engineering (ASE)*, pages 199–208. IEEE, 2006.

[22] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace. The design of the zinc modelling language. *Constraints*, 13(3):229–267, 2008.

[23] I. Neamtiu and T. Dumitras. Cloud software upgrades: Challenges and opportunities. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2011 International Workshop on the*, pages 1–10, Sept. 2011.

[24] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming (CP)*, pages 529–543, 2007.

[25] Normation. Rudder – open source automation & compliance. `https://www.rudder-project.org/`. Retrieved April 2014.

[26] Opscode. Chef. `http://www.opscode.com/chef/`. Retrieved October 2013.

[27] Puppet Labs. Marionette collective. `http://docs.puppetlabs.com/mcollective/`. Retrieved October 2013.

[28] RedHat. Augeas – a configuration API. `http://augeas.net/`. Retrieved April 2014.

[29] C. Schulte, M. Lagerkvist, and G. Tack. Gecode. `http://www.gecode.org/`. Retrieved October 2013.

[30] P. J. Stuckey, M. G. de la Banda, M. Maher, J. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *International Conference on Logic Programming (ICLP)*, volume 3668 of *LNCS*, pages 9–13, 2005.

[31] S. van der Burg, E. Dolstra, and M. de Jonge. Atomic upgrading of distributed systems. In *Hot Topics in Software Upgrades*, pages 1–5. ACM, 2008.

[32] VMWare. Cloud Foundry, deploy & scale your applications in seconds. Retrieved October 2013, `http://www.cloudfoundry.com/`.