Optimal Provisioning in the Cloud. Technical report of the Aeolus project

Roberto Di Cosmo, Michaël Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski

Optimal Provisioning in the Cloud*

Technical Report and Proofs

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
zwolakowski@pps.univ-paris-diderot.fr

01/03/2013

Abstract

Complex distributed systems are classically assembled by deploying several existing software components to multiple servers. Building such systems is a challenging problem that requires a significant amount of problem solving as one must i) ensure that all inter-component dependencies are satisfied; ii) ensure that no conflicting components are deployed on the same machine; and iii) take into account replication and distribution to account for quality of service, or possible failure of some services.

We propose a tool, *Zephyrus*, that automates to a great extent assembling complex distributed systems. Given i) a high level specification of the desired system architecture, ii) the set of available components and their requirements) and iii) the current state of the system, Zephyrus is able to generate a formal representation of the desired system, to place the components in an optimal manner on the available machines, and to interconnect them as needed.

1 Introduction

In contrast to classic, monolithic software that runs locally on one 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. Designing and running such systems is a complex task, far different from classic software management: it is like building a puzzle (each running service being a piece) where you only

*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

know one part of the picture (the expected functionality). More precisely, the system designer must solve the following problems: i) choose which services to use and how to configure them, knowing that services may depend on (and/or be in conflict with) each other; ii) consider fault tolerance and quality of service issues, and provide enough instances of each service to deal with that; iii) 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; iv) choose which implementation of each service to install on which location, knowing that implementations (or *packages*), like services, have dependencies and conflicts; and v) install each package and start each service on the chosen architecture. Also, it is possible that the architecture on which to install the system is not initially empty, maybe because the new system is an upgrade of an existing one that will get replaced, or because the designer have to co-host the new system with another one to decrease the cost. In that case, one might want to design the system to reuse parts of the existing one, to get a more efficient installation process. This adds yet another layer of complexity to the design process.

To lower complexity, many industrial initiatives develop tools [VMW, Can] that allow to select, configure, and push to a "cloud" some well defined services, thus reducing application development cost. However, these tools are only useful once the puzzle is finished, 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 the puzzle currently requires a significant amount of manual intervention, so that in practice large software stacks are often managed using custom scripts and manual techniques, which are error prone and fragile [ND11].

The goal of our work is to provide a generic, automatic and sound alternative to these scripts and techniques. In this paper we provide a first big step towards that goal: a tool, called Zephyrus,[1] that automatically generates an abstract representation (or *configuration*) of the expected system. More precisely Zephyrus takes as input: i) a specification of the system's expected functionalities; ii) the set of available services, which can serve as building blocks, with their requirements, replication policies and resource consumption; iii) informations concerning the implementation of the services (e.g. the `apache` service is provided by the `www-servers/apache` package on `Gentoo` linux, by the `apache2` package on `Debian`, etc); and iv) the (possibly non-empty) architecture (or *initial configuration*) on which the system will be installed. Moreover, the system designer can choose one out of a set of optimization criteria that capture preferences like "use the smallest number of locations", or "modify as little as possible the pre-existing configuration". From such an input, Zephyrus generates a precise description of which packages must be installed on which location, which services must be started and how they must be linked together.

[1] Zephyrus is free software, implemented in OCaml, and available at `www.mancoosi.org/software/zephyrus/`

The generation process works in three steps. First, all of Zephyrus inputs are translated to constraints on positive integers whose goal is to represent packages, services, and locations requirements. This ability to uniformly capture all of it with integer constraints is the cornerstone of our approach: it allows to deal with the many facets of a system design as a whole, and thus ensure the completeness of our algorithm and the optimality of the generated configuration. Second, an optimal solution is provided by an external constraint solver. This solution specifies which packages must be installed where and how many instances of a service must be started on each location. And third, from the given solution and the initial configuration, we generate the final configuration, reusing as many existing running services as possible.

The basic usage of Zephyrus is thus the generation of a configuration from an input specification. Due to its expressiveness and its capacity to take as input non-empty configurations, Zephyrus's usage covers several very useful scenarios: i) running Zephyrus on a broken system will generate a configuration that fixes the problem; ii) similarily, Zephyrus can be used to update a system to use new services or new replication policies; and iii) without an input configuration, Zephyrus will generate a configuration using as many locations and resources as necessary, thus giving an estimate of the architecture required to deploy the expected system.

Finally, our goal is to provide a sound tool, and thus all of the elements used in our approach are formalized. Our *component model*, inspired by Aeolus [DCZZ12], encodes each service with its replication policy by a *component type* using *ports* tagged with an *arity* to encode requirements, provides and conflicts. Packages and repositories are abstracted with a model close to [MBC+06]. Our model for configurations is based on Aeolus [DCZZ12], but extended to take locations, repositories, packages and resources into account. Finally, our notion of specifications is entirely new, and is presented with a formal syntax and semantics which defines when a configuration satisfies a specification. Based on this formalization, Zephyrus is proven complete and correct: it will always find a configuration that is optimal w.r.t. the chosen criterion if one exists; the generated configuration does provide the expected functionalities, and abides by the constraints defined by the replication policies, the dependencies and conflicts between services, etc.

This paper is organized as follows. Section 2 shows Zephyrus at work on a realistic use case; Section 3 introduces the formal definitions of components, packages and repositories, configurations and specifications; Section 4 shows how to encode a system design problem into numerical constraints; Section 5 presents the generation of a configuration from a solution of the constraints; before concluding, Section 6 compares our contribution to related works.

2 Approach

In this Section, we present the different usages of Zephyrus.

2.1 Basic Usage: Configuration generation

Use case. Let consider we have to deploy the popular blog platform Wordpress on some public cloud setting (e.g. Amazon EC2, Windows Azure, ...). In addition to being a realistic use case, this 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 DBMS instance, more precisely an instance of MySQL, 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 — i.e. those meant to sustain high visit loads and be resilient to machine failures — are usually more complex than that 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 a 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 different 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. Various kinds of design constraints should be taken into account when planning such a complex system. Some of these constraints come from package providers and cannot be changed. For example, Wordpress, Varnish, etc. usually come from distribution packages and have their own set of dependencies and conflicts which must be respected when installing the software on each machine.

On the other hand, "house" requirements are defined by the designers to capture some ad-hoc policy. For example, designers might want:

- 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 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 2 Wordpress instances;
- no more than 1 DNS server deployed in the administrative domain;
- or, again, that different Wordpress (and MySQL) instances are deployed on different locations.[2]

[2] it is technically possible to co-locate multiple, say, MySQL instances on the same machine,

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 these constraints are not intrinsically related to the software components we are using, but are rather an encoding of explicit architectural choices.

Architecture. Zephyrus consumes as input (1) a description of all the existing constraints, which come in various formats due to their different origins (e.g. package database, designer choices, physical resources of machines, etc.); this is called a *universe*. Additionally, it takes a (2) description of the current system *configuration* (which machine exist, what is currently deployed where, etc.) and a (3) *specification* characterizing the system that architects would like to achieve. As part of the specification, the architects can also specify 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).

Internally, Zephyrus translates all the constraints into a coherent whole, as described in Section 4. Once assembled, the constraints are passed to an external constraint solver[3] that computes the number of both service instances and their interconnections (called *bindings*) that are needed to obtain the desired state, while optimizing for the specified objective function. As the aggregate number of instances and bindings alone is not sufficient for deployment, Zephyrus then produces an actual configuration by allocating services to machines and by computing how they should be connected. This is done using the algorithms described in Section 5.

Using Zephyrus. Figure 1 shows the application of our approach 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 Aeolus [DCZZ12], each one with its own requirements, conflicts, and house policy. For instance, the HTTP load balancer requires 3 Wordpress replicas, whereas the DNS load balancer requires 7 and sports a conflict on other DNS services, as per house 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. Additionally, Zephyrus takes in input the *implementation* relation that maps each service to the set of packages that im-

but it would be pointless to do so when we are seeking fault tolerance and load balancing.

[3]currently, Zephyrus uses the FaCiLe constraint solver library for this step `http://www.recherche.enac.fr/log/facile/`. However, given the solver is used as a black-box, it is possible to use other solver components in its stead.

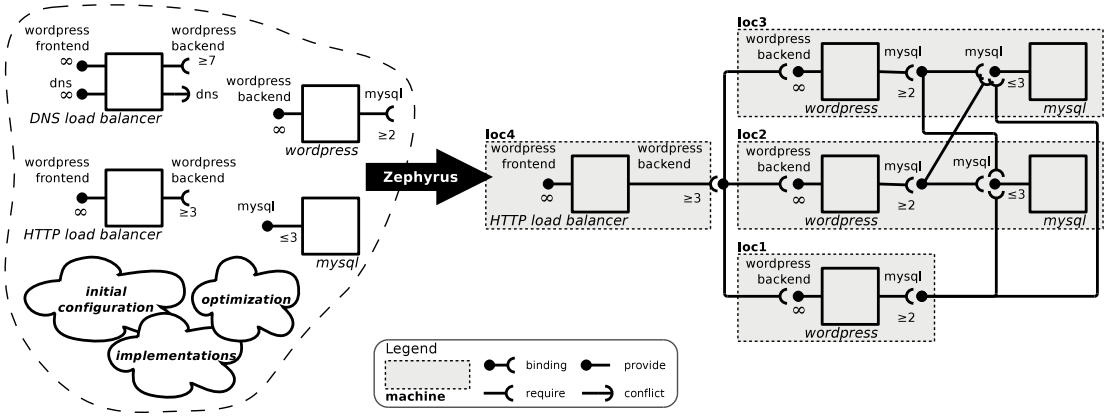


Figure 1: Zephyrus usage to design a scalable, fault-tolerant Wordpress deployment

plements it. These two parts of the universe is given in input to Zephyrus as the following JSON file:

```
{ "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" : [["@mysql", 2]],
    "consume" : [["ram", 512]] },
  { "name"   : "MySQL",
    "provide" : [["@mysql", 3]],
    "consume" : [["ram", 512]] } ],
  "implementation": [
    [ "DNS-load-balancer", ["bind9"] ],
    [ "HTTP-load-balancer", ["varnish"] ],
    [ "Wordpress", ["wordpress"] ],
    [ "MySQL", ["mysql-server"] ] ]
}
```

The former part of it describes component types and their requirements; the latter the distribution packages that should be installed to realize the services on actual machines. Because of its size, the rest of the universe (repositories and packages) is not included in this file, but as an annex zip file. Moreover, this file

can first be processed by coinst [CV11] that abstracts packages into dependencies equivalent classes, reducing largely the number of packages Zephyrus needs to process (the Debian Squeeze repository contains ≈30'000 packages).

In our example, we start with an initial configuration consisting of 6 bare locations with 2Go of RAM. Such configuration is given in input to Zephyrus as the following JSON file (excerpt):

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

Finally, Zephyrus needs as input a specification of the desired target state:

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

This specification asks for exactly one Wordpress frontend (more precisely exactly one service offering a `wordpress-frontend` port) and imposes that no machine is deployed with more than one instance of either MySQL/Wordpress services on it. Note that no constraint is imposed on the co-location of *different* services on the same machine.

Equipped with all this, we are now ready to ask Zephyrus to compute the final configuration:

```
$ zephyrus -repo debian-squeeze Packages.coinst \
    -u univ-1.json -ic conf-1.json -spec spec-1.spec \
    -opt compact
```

In addition to the obvious ones (universe, configuration, specification), we pass two extra parameters to Zephyrus. `-repo` is the zip file containing the informations about repositories and packages. The other parameter, `-opt`, is used to request the optimization w.r.t. a specific objective function. Currently, one must choose among a limited set of objective functions. Here, `compact` is used to request the minimization of the number of needed (i.e. non empty) machines (see Section 4 for the formal definition).

The actual output of Zephyrus is too verbose to be listed here in full, so we only provide some excerpt from it. The format is the same as for configurations, and starts with location descriptions (excerpt):

```
{ "locations": [
    { "name": "loc1",
      "provide_resources": [ [ "ram", 2048 ] ],
      "repository": "debian-squeeze",
      "packages_installed": [ "wordpress (= 3.3.2-1)",
        "libgd2-xpm (x 125)" ] },
    { "name": "loc2",
```

```
    "provide_resources": [ [ "ram", 2048 ] ],
    "repository": "debian-squeeze",
    "packages_installed": ["mysql-server (= 5.1.49-3)",
      "2vcard (x 23886)", "wordpress (= 3.3.2-1)",
      "libgd2-xpm (x 125)" ] },
```

We can see that each location is associated with a list of packages installed there. The second part of the configuration, not shown in the initial configuration because it was empty, is the list of service instances mapped to their deployment locations (excerpt):

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

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

```
"bindings": [
  { "port": "@wordpress-backend",
    "requirer": "loc4-HTTP-load-balancer-1",
    "provider": "loc3-Wordpress-1" },
  { "port": "@wordpress-backend",
    "requirer": "loc4-HTTP-load-balancer-1",
    "provider": "loc2-Wordpress-1" },
  { "port": "@mysql",
    "requirer": "loc1-Wordpress-1",
    "provider": "loc2-MySQL-1" }
```

The complete result is shown on the right of Figure 1, where shaded boxes denote locations. All choices there (load balancer solution, mapping of service instances to machines, bindings, etc.) has been made by Zephyrus. Note how services have been co-located, where possible, to minimize the number of used machines (4 out of the 6 machines that were available). The obtained solution is optimal w.r.t. the desired metric.

2.2 Fixing a Configuration

Let suppose given an existing installation of the Wordpress system like the one computed in Section 2.1, except that the main Wordpress services wasn't configured to use the different backends. This installation typically cannot function properly, as all attempts to access a page of the website will end up in an error 404. Running Zephyruswith such configuration in input and with the `conservative` optimization option (that will keep the installed services) will see that Wordpress wasn't properly configured, and create the bindings to end up in a valid configuration, namely the one in Section 2.1.

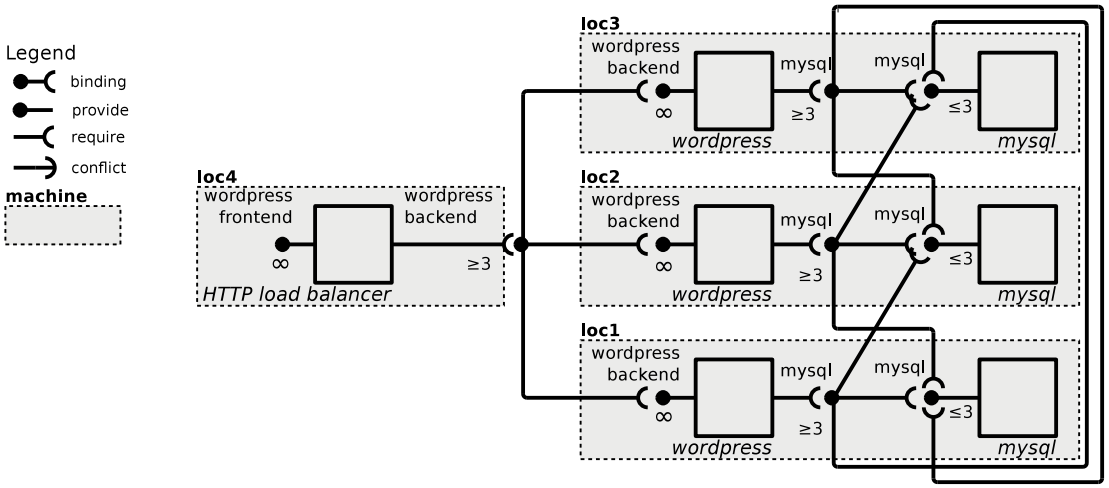


Figure 2: The configuration, after updating the redundancy requirements and applying Zephyrus

2.3 Updating Services and Replication Policies

Let consider given an existing installation of the Wordpress system as computed in Section 2.1: one might want to increase the redundancy at the MySQL level, by increasing the amount of minimum MySQL entry points for each Wordpress instance from 2 to 3. By rerunning Zephyrus on the given system, after modifying the universe to reflect the extra redundancy, we have obtained a new configuration, presented in Figure 2 where no extra machines are spawn, but a new MySQL service instance is added on location `loc1`.

2.4 Configuration Estimation

Let suppose that in Section 2.1, we didn't have an initial configuration, but we wanted to know more or less how many locations were necessary to host the application. Running Zephyrus without an initial configuration will generate the result configuration in two steps. First, it computes how many services instances are necessary to run the application by generating and solving the constraints without the part about locations. This number gives us a first estimate of the number of needed locations, as at most one location per service is needed. In our case, Zephyrus finds that 6 services are needed to deploy Wordpress, and thus generates a first estimate of 6 locations. Then, Zephyrus performs a second pass, with these 6 locations as input. This second pass ends up with a configuration similar to the one computed in Section 2.1, but because the generated locations do not have any limit on the resources they provide, the load-balancer is put together with a backend and a database: only 3 locations are used, with one location requiring 3052Mo of RAM.

3 Formal Model

In this section, we formally define the different elements (services, configurations, packages, etc) used in Zephyrus. As mentioned before, this formalization abstracts services and replication policy by the notion of component types. Moreover, to follow the vocabulary of [DCZZ12], a running instance of a service is called a *component*. We structure our presentation into four parts: i) a *Universe* declares the different component types, repositories and packages that we can use to build a configuration; ii) a *Configuration* models a system (i.e. a set of components bound together) with its underlying architecture (i.e. a set of locations hosting the components, with the packages that implement them); ii) a *Specification* states the required features of the final configuration; and iv) an *Optimization Function* allows to select out of a set of possible configurations, the optimal one. Before giving a formal definition of these elements, we first introduce several infinite and disjoint sets and the notion of *mapping* on which we base our definitions. In the following, we suppose given a set of component type names $\mathcal{T}$, ranged over by t_1, t_2, etc; a set of port names $\mathcal{P}$, ranged over by p_1, p_2, etc; a set of component names $\mathcal{C}$, ranged over by c_1, c_2, etc; a set of package names $\mathcal{K}$, ranged over by k_1, k_2, etc; a set of repository names $\mathcal{D}$, ranged over by r_1, r_2, etc; a set of location names $\mathcal{L}$, ranged over by l_1, l_2, etc; and a finite set of resource names $\mathcal{O}$, ranged over by o_1, o_2, etc. Also, given two sets E and F, a *mapping* $f : E \mapsto F$ is function f whose domain $dom(f)$ is a finite subset of E, and whose image is included in F.

3.1 Universe

A universe declares the different component types and repositories we can use to build a configuration. *Component types* are services, with dependencies and conflicts modeled by provided, required and conflicting *ports*, together with their replication policy, modeled by a *maximal output arity* on provided ports, and *minimal input arity* on required ports.

Definition 1 (Component Type)**.** *A* component type $\mathcal{J}$ *is a 4-ple* $\langle \mathbf{P}, \mathbf{R}, \mathbf{C}, \mathbf{f} \rangle$ *where:*

- $\mathbf{P} : \mathcal{P} \mapsto \mathbb{N}^+ \cup \{\infty\}$ *is a mapping defining the provided ports of the component type, with their arity;*
- $\mathbf{R} : \mathcal{P} \mapsto \mathbb{N}^+$ *is a mapping defining the required ports of the component type, with their arity;*
- $\mathbf{C} \subset \mathcal{P}$ *is the finite set of ports the resource type is in conflict with;*
- $\mathbf{f} : \mathcal{O} \to \mathbb{N}$ *is a function stating how much of each resource this component type consumes.*

We note Γ *the set of component types.*

On the other hand *Packages*, provided by some *repositories*, implement the component types. Unlike components who may provide, depend on, or conflict with ports, packages depend on or conflict with other packages.

Definition 2 (Package and Repository)**.** *A* package $\mathcal{H}$ *is a triple* $\langle \mathbf{R}, \mathbf{C}, \mathbf{f} \rangle$ *where*

- $\mathbf{R} \subset P(\mathcal{K})^4$ *is the set of dependencies of the package: for each set* $\{k_i\} \in \mathbf{R}$ *at least one* k_i *must be installed for the current package to be installed as well;*
- $\mathbf{C} \subset \mathcal{K}$ *is the set of packages the current one is in conflict with;*
- $\mathbf{f} : \mathcal{O} \to \mathbb{N}$ *is a function stating how much of each resource this package consumes.*

We note Π *the set of all packages. A* repository $\mathcal{R}$ *is a mapping from package names* $\mathcal{K}$ *to packages. We note* Ω *the set of all repositories.*

Finally, we can formally define what an universe is.

Definition 3 (Universe)**.** *A* universe $\mathcal{U}$ *is a triple* $\langle \mathbf{N}, \mathbf{I}, \mathbf{Y} \rangle$ *where*

- $\mathbf{N} : \mathcal{T} \mapsto \Gamma$ *is a finite mapping defining the set of component types with their names;*
- $\mathbf{I} \subset dom(\mathbf{N}) \times \mathcal{K}$ *is the implementation relation;*
- $\mathbf{Y} : \mathcal{D} \mapsto \Omega$ *is a mapping defining the available repositories with their names.*

To simplify our presentation, we suppose that the repositories of a universe $\mathcal{U}$ *all have distinct domains.*

Notation. Given a universe $\mathcal{U} = \langle \mathbf{N}, \mathbf{I}, \mathbf{Y} \rangle$, we note: $\mathcal{U}_{dt}$ the set of component type names of $\mathcal{U}$; $\mathcal{U}_{dp}$ the set of ports used in $\mathcal{U}$; $\mathcal{U}_{dr}$ the set of repository names of $\mathcal{U}$; $\mathcal{U}_{dk}$ the set of all pacakages names in $\mathcal{U}$. Moreover, $\mathcal{U}_i : \mathcal{U}_{dt} \mapsto P(\mathcal{U}_{dk})$ gives the set of packages implementing each component type in $\mathcal{U}$; $\mathcal{U}_w : \mathcal{U}_{dk} \mapsto \Pi$ gives the packages of all package names in $\mathcal{U}$; $\mathcal{U}\mathtt{R} : \mathcal{P} \to P(\mathcal{U}_{dt})$ gives the set of component types that require the port in parameter; $\mathcal{U}\mathtt{P} : \mathcal{P} \to P(\mathcal{U}_{dt})$ gives the set of types that provide the port in parameter; and $\mathcal{U}\mathtt{P} : \mathcal{P} \to P(\mathcal{U}_{dt})$ gives the set of types that conflict with the port in parameter. Also, given a component type name t, we note $\mathcal{U}(t)$ for $\mathbf{N}(t)$, given a repository name r, we note $\mathcal{U}(r)$ for $\mathbf{Y}(r)$, and given a package name k, we note $\mathcal{U}(k)$ for $\mathcal{U}_w(k)$. Formally, we have

$$\mathcal{U}_{dt} \triangleq dom(\mathbf{N}) \qquad \mathcal{U}_{dr} \triangleq dom(\mathbf{Y}) \qquad \mathcal{U}_i(t) \triangleq \{k \mid (t,k) \in \mathbf{I}\} \qquad \mathcal{U}_w \triangleq \bigcup_{r \in \mathcal{U}_{dr}} \mathcal{U}(r)$$

$$\mathcal{U}_{dk} \triangleq \bigcup_{r \in \mathcal{U}_{dk}} dom(\mathcal{U}(r)) \qquad \mathcal{U}_{dp} \triangleq \bigcup_{t \in \mathcal{U}_{dt}} (\mathcal{U}(t).\mathbf{C} \cup dom(\mathcal{U}(t).\mathbf{R}) \cup dom(\mathcal{U}(t).\mathbf{P}))$$

$$\mathcal{U}\mathtt{R}(p) = \{t \mid t \in \mathcal{U}_{dt} \wedge p \in dom(\mathcal{U}(t).\mathbf{R})\} \qquad \mathcal{U}\mathtt{P}(p) = \{t \mid t \in \mathcal{U}_{dt} \wedge p \in dom(\mathcal{U}(t).\mathbf{P})\} \qquad \mathcal{U}\mathtt{C}(p) = \{t \mid t \in \mathcal{U}_{dt} \wedge p \in \mathcal{U}(t).\mathbf{C}\}$$

Given a tuple $T = \langle \ell_1, \ldots \ell_i \rangle$, we note $T.\ell_i$ the lookup operation that retrieves the element ℓ_i from the tuple. For instance, $\mathcal{U}(t).\mathbf{R}(p)$ stands for the minimum arity required by the component type t for the port p.

[4] We write $P(X)$ for the set of subsets of X

3.2 Configuration

A configuration C is given by a set of locations with their characteristics (how many resources they provide, what repository and packages are installed), a set of components and their bindings.

Definition 4 (Configuration)**.** *A* configuration C *is a triple* $\langle L, W, B\rangle$ *where*

- L *is a mapping from* $\mathcal{L}$ *to triples* $\langle \phi, r, M\rangle$ *where* $\phi : \mathcal{O} \to \mathbb{N}$ *is a function stating how many resources this location provides;* $r \in \mathcal{D}$ *is the name of the repository installed on that location;* $M \subset \mathcal{K}$ *is the set of packages installed on that location;*
- W *is a mapping from* $\mathcal{C}$ *to pairs* $\langle l, t\rangle$ *where* $l \in dom(L)$ *and* $t \in \mathcal{T}$*, stating for each component its location and its type;*
- $B \subset \mathcal{K} \times dom(W) \times dom(W)$ *is the set of* bindings*, namely 3-ple composed by a port, the component that requires that port and the component that provides it.*

Notation. Given an configuration $C = \langle L, W, B\rangle$, we note C_l (resp. C_c, C_t, C_k) the set of locations (resp. components, component types, package) in that configuration. Moreover, given a location l, a component c, a component type name t and a package name k, we note: $C(l)$ for $L(l)$; $C(c)$ for $W(c)$; $C.\texttt{type}(c)$ the type of c; $C(l,t)$ the set of components that are placed on l and whose type is t; and $C(l,k)$ the boolean stating whether the package k is installed on l. Formally, we have

$$C_l \triangleq dom(L) \quad C_c \triangleq dom(W) \quad C_t \triangleq \{t \mid \exists c \in C_c, \exists l \in C_l,\, W(c) = (l,t)\} \quad C_k \triangleq \bigcup_{l \in C_l} C(l).M$$

$$C(l,t) \triangleq \{c \mid c \in dom(W) \wedge W(c) = (l,t)\} \quad C(l,k) \triangleq (k \in L(l).M)$$

An important notion on configuration is the *correctness* w.r.t. a universe. Configurations have to respect the constraints given by the input universe: to only use the elements (e.g. component types, packages) declared in the input universe, and to use them right (e.g. all component must be implemented by a package, all requirements must be fulfilled by the right number of connections). We structure our definition of correctness in three parts: component types, packages, and resources.

Definition 5. *Suppose given a configuration* $C = \langle L, W, B\rangle$ *and a universe* $\mathcal{U} = \langle \mathbf{N}, \mathbf{I}, \mathbf{Y}\rangle$*.* C *is* component-valid *w.r.t.* $\mathcal{U}$ *if for all* $c \in C_c$*, the pair* $\langle l, t\rangle = W(c)$ *is such that* $t \in \mathcal{U}_{dt}$ *and:*

$$\begin{cases} \forall p \in \mathcal{P} \setminus dom(\mathcal{U}(t).\mathbf{P}),\ \{c' \mid (p, c', c) \in B\} = \emptyset \\ \forall p \in dom(\mathcal{U}(t).\mathbf{P}),\ \#\{c' \mid (p, c', c) \in B\} \leq \mathcal{U}(t).\mathbf{P}(p) \end{cases} \tag{1}$$

$$\forall p \in dom(\mathcal{U}(t).\mathbf{R}),\ \#\{c' \mid (p, c, c') \in B\} \geq \mathcal{U}(t).\mathbf{R}(p) \tag{2}$$

$$\forall p \in \mathcal{U}(t).\mathbf{C},\ \forall c' \in C_c \setminus \{c\}, C.\texttt{type}(c') \notin \mathcal{U}_{\mathbf{P}}(p) \tag{3}$$

$$\exists (t,k) \in \mathbf{I},\ k \in L(l).M \tag{4}$$

Basically, these formula means that: the components are not bound to too many clients (equation (1)); all the requirements of all components are satisfied (equation (2)); there are no conflicts (equation (3)); and all components are implemented by a package (equation (4)).

Definition 6. *Suppose given a configuration* $C = \langle L, W, B\rangle$ *and a universe* $\mathcal{U} = \langle \mathbf{N}, \mathbf{I}, \mathbf{Y}\rangle$. *$C$ is* package-valid *w.r.t. $\mathcal{U}$ if for all $l \in dom(L)$, the triple $\langle \phi, r, M\rangle = L(l)$ is such that $r \in \mathcal{U}_{dr}$ and:*

$$M \subset \mathcal{U}(r) \tag{5}$$
$$\forall k \in M,\, \exists m \in \mathcal{U}(k).\mathbf{R},\, m \subset M \tag{6}$$
$$\forall k \in M,\, \mathcal{U}(k).\mathbf{C} \cap M = \emptyset \tag{7}$$

Basically, these formula means that: all packages are declared in $\mathcal{U}$, in the right repository (equation (5)); all the dependencies of all packages are satisfied (equation (6)); and there are no conflicts (equation (7)).

Definition 7. *Suppose given a configuration* $C = \langle L, W, B\rangle$ *and a universe* $\mathcal{U} = \langle \mathbf{N}, \mathbf{I}, \mathbf{Y}\rangle$. *$C$ is* resource-valid *w.r.t. $\mathcal{U}$ if for all locations $l \in dom(L)$ and all resources $o \in \mathcal{O}$, the following inequality holds:*

$$\sum_{t \in \mathcal{U}_{dt}} \#(C(l,t)) \times \mathcal{U}(t).\mathbf{f}(o) + \sum_{p \in L(l).M \cap \mathcal{U}_{dk}} \mathcal{U}(p).\mathbf{f}(p) \leq L(l).\phi(o)$$

Definition 8. *A configuration C is* valid *w.r.t. a universe $\mathcal{U}$ (noted $\mathcal{U} \vdash C$) iff it is component-, package- and resource-valid w.r.t. $\mathcal{U}$.*

3.3 Specifications

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 operations. Intuitively, these basic constraints specify how many elements (packages, component types, etc) are in the generated configuration, using terms of the form $\#\ell$ that correspond to the number of instances of the element ℓ in the system. For instance, it is possible to state that we want at least three instances of the component type `apache`: "`#apache` $\geq$ `3`" with `#apache` representing the number of instance of `apache` in the configuration. Moreover, it is also possible to have 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; and S_l is a constraint specifying what is the contents of the machine (basically, S_l is S without locations). For instance, we can specify that we want exactly one location with `redhat` installed and `apache` running: "`#(_){redhat: apache` $\geq$ `1} = 1`". Finally, for flexibility, it is possible to use global variables (noted X) in specifications.

The following definition formally presents the semantics of a specification:

Table 1 Specification Syntax

S	::=	$\mathbf{true} \mid e\ op\ e$	Specification
	$\mid$	$S \wedge S \mid S \vee S$	
	$\mid$	$S \Rightarrow S \mid \neg S$	
e	::=	$X \mid n \mid \#\ell$	Expression
	$\mid$	$e + e \mid e - e \mid n \times e$	
ℓ	::=	$k \mid t \mid p$	Elements
	$\mid$	$(J_\phi)\{J_r : S_l\}$	
S_l	::=	$\mathbf{true} \mid e_l\ op\ e_l$	Local Specification
	$\mid$	$S_l \wedge S_l \mid S_l \vee S_l$	
	$\mid$	$S_l \Rightarrow S_l \mid \neg S_l$	
e_l	::=	$X \mid n \mid \#\ell_l$	Local Expression
	$\mid$	$e_l + e_l \mid e_l - e_l \mid n \times e_l$	
ℓ_l	::=	$k \mid t \mid p$	Local Elements
J_ϕ	::=	$\_ \mid o\ op\ n; J_\phi$	Resource Constraint
J_r	::=	$r \mid r \vee J_r$	Repository Constraint
op	::=	$\leq \mid = \mid \geq$	Operators

Definition 9. *Suppose given a specification S: $fv(S)$ stands for the set of variables used in S. Given a universe $\mathcal{U}$, a configuration C validates the specification S (noted $C \vdash S$) if there exists a function σ from $fv(S)$ to integers such that $C, \sigma \vdash S$ can be derived from the rules presented in Tables 2 and 3.*

Basically, these tables maps the different elements of the configuration (locations, components, packages) to the different elements $\#\ell$ in the specification, and ensures that the function σ, extended with this mapping, is a solution for S.

3.4 Optimization Function

The last piece of input for our tool is the optimization function $\mathcal{F}$ that allows us to select the optimal configuration among all configurations validating the specification. We consider here only three kind of optimizations, just to give an idea of the expressiveness of our approach: `compact` selects the solution that uses the least locations; `spread` selects the solution that uses the least components and the most locations, to improve load distribution; `conservative` selects the solution that is the closest to the initial state of the system.

4 Translation into Constraints

We now present the translation of the various inputs into *numerical constraints* plus one function (for the optimization). Basically, we use numeric variables to represent important informations about a configuration: for instance, we note $N(l, t)$ the variable corresponding to the number of instances of the component

Table 2 Specification Validation (1/2)

$$\text{SV:True}\quad C,\sigma \vdash \mathbf{true}$$

$$\text{SV:Exp}\quad \frac{C,\sigma \vdash e \Rightarrow n \qquad C,\sigma \vdash e' \Rightarrow n' \quad n\ op\ n'}{C,\sigma \vdash e\ op\ e'}$$

$$\text{SV:And}\quad \frac{C,\sigma \vdash S_1 \quad C,\sigma \vdash S_2}{C,\sigma \vdash S_1 \wedge S_2}$$

$$\text{SV:Not}\quad \frac{C,\sigma \nvdash S}{C,\sigma \vdash \neg S}$$

$$\text{SV:Or1}\quad \frac{C,\sigma \vdash S_1}{C,\sigma \vdash S_1 \vee S_2}$$

$$\text{SV:Or2}\quad \frac{C,\sigma \vdash S_2}{C,\sigma \vdash S_1 \vee S_2}$$

$$\text{SV:Imply1}\quad \frac{C,\sigma \vdash S_1 \qquad C,\sigma \vdash S_2}{C,\sigma \vdash S_1 \Rightarrow S_2}$$

$$\text{SV:Imply2}\quad \frac{C,\sigma \nvdash S_1}{C,\sigma \vdash S_1 \Rightarrow S_2}$$

$$\text{SV:Var}\quad C,\sigma \vdash X \Rightarrow \sigma(X)$$

$$\text{SV:Number}\quad C,\sigma \vdash n \Rightarrow n$$

$$\text{SV:Package}\quad C,\sigma \vdash k \Rightarrow \sum_{l \in C_l} \#(C(l).M \cap \{k\})$$

$$\text{SV:Type}\quad C,\sigma \vdash t \Rightarrow \sum_{l \in C_l} \#(C(l,t))$$

$$\text{SV:Port}\quad C,\sigma \vdash p \Rightarrow \sum_{l \in C_l} \sum_{t \in \mathcal{U}^{\mathbf{P}}(p)} \#(C(l,t)) \times \mathcal{U}(t).\mathbf{P}(p)$$

$$\text{SV:Plus}\quad \frac{C,\sigma \vdash e_1 \Rightarrow n_1 \qquad C,\sigma \vdash e_2 \Rightarrow n_2}{C,\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

$$\text{SV:Minus}\quad \frac{C,\sigma \vdash e_1 \Rightarrow n_1 \qquad C,\sigma \vdash e_2 \Rightarrow n_2}{C,\sigma \vdash e_1 - e_2 \Rightarrow n_1 - n_2}$$

$$\text{SV:Times}\quad \frac{C,\sigma \vdash e \Rightarrow n}{C,\sigma \vdash n' \times e \Rightarrow n' \times n}$$

$$\text{SV:Loc}\quad \frac{v = \{l \in C_l \mid C,l \vDash J_\phi \wedge C,l \vDash J_r \wedge C,\sigma,l \vDash S_l\}}{C,\sigma \vdash (J_\phi)\{J_r : S_l\} \Rightarrow \#v}$$

type t on the location l. The numerical constraints built on these variables ensure that the design constraints from the input universe and the input specification are satisfied. For instance, we have constraints that ensure that all requests are satisfied by some provides, or that all installed components are implemented by an package installed on the same location. We then use an external solver to solve the generated constraints. Using the optimization function, the solver computes an optimal solution to the problem, computing the number of instance of each component types, which repository, and which packages must be installed on each location.

4.1 Numerical Constraints

Table 4 presents the syntax of constraints. Basically, a constraint $\mathcal{A}$ is a set of comparisons between numerical expressions $u\ op\ u$, combined using the logic operators $\wedge$, $\vee$, $\Rightarrow$ and $\neg$. Expressions u are numerical expressions, with positive integers n, variables X, addition, substraction and multiplication with a integer, extended with: i) a set of specific variables representing a configuration; and ii) reified constraints $\|\mathcal{A}\|$, whose value is 1 if $\mathcal{A}$ is true, 0 otherwise. The semantics of the extra variables is as follow:

- $N(\ell_l)$ is the number of instances of ℓ_l (component types, ports and packages) installed globally in the configuration;
- $N(l, \ell_l)$ is the number of instances of ℓ_l (component types, ports and

Table 3 Specification Validation (2/2)

$$\text{SV:Res1}\quad C, l \vDash \_$$

$$\text{SV:Res2}\quad \frac{C, l \vDash J_\phi \quad C(l).\phi(o)\ op\ n}{C, l \vDash o\ op\ n; J_\phi}$$

$$\text{SV:Rep1}\quad \frac{C(l).r = r}{C, l \vDash r}$$

$$\text{SV:Rep2}\quad \frac{C(l).r = r}{C, l \vDash r \vee J_r}$$

$$\text{SV:Rep3}\quad \frac{C, l \vDash J_r}{C, l \vDash r \vee J_r}$$

$$\text{SV:L:True}\quad C, \sigma, l \vDash \mathbf{true}$$

$$\text{SV:L:Exp}\quad \frac{C, \sigma, l \vDash e_l \Rightarrow n \quad C, \sigma, l \vDash e'_l \Rightarrow n' \quad n\ op\ n'}{C, \sigma, l \vDash e_l\ op\ e'_l}$$

$$\text{SV:L:And}\quad \frac{C, \sigma, l \vDash S_l \quad C, \sigma, l \vDash S'_l}{C, \sigma, l \vDash S_l \wedge S'_l}$$

$$\text{SV:L:Not}\quad \frac{C, \sigma, l \nvDash S_l}{C, \sigma, l \vDash \neg S_l}$$

$$\text{SV:L:Or1}\quad \frac{C, \sigma, l \vDash S_l}{C, \sigma, l \vDash S_l \vee S'_l}$$

$$\text{SV:L:Or2}\quad \frac{C, \sigma, l \vDash S'_l}{C, \sigma, l \vDash S_1 \vee S'_l}$$

$$\text{SV:L:Imply1}\quad \frac{C, \sigma, l \vDash S_l \quad C, \sigma, l \vDash S'_l}{C, \sigma, l \vDash S_l \Rightarrow S'_l}$$

$$\text{SV:L:Imply2}\quad \frac{C, \sigma, l \nvDash S_l}{C, \sigma, l \vDash S_l \Rightarrow S'_l}$$

$$\text{SV:L:Var}\quad C, \sigma, l \vDash X \Rightarrow \sigma(X)$$

$$\text{SV:L:Number}\quad C, \sigma, l \vDash n \Rightarrow n$$

$$\text{SV:L:Package}\quad C, \sigma, l \vDash k \Rightarrow \#(C(l).M \cap \{k\})$$

$$\text{SV:L:Type}\quad C, \sigma, l \vDash t \Rightarrow \#(C(l, t))$$

$$\text{SV:L:Port}\quad C, \sigma, l \vDash p \Rightarrow \sum_{t \in \mathcal{U}\mathbf{P}(p)} \#(C(l, t)) \times \mathcal{U}(t).\mathbf{P}(p)$$

$$\text{SV:L:Plus}\quad \frac{C, \sigma, l \vDash e_l \Rightarrow n_1 \quad C, \sigma, l \vDash e'_l \Rightarrow n_2}{C, \sigma, l \vDash e_l + e'_l \Rightarrow n_1 + n_2}$$

$$\text{SV:L:Minus}\quad \frac{C, \sigma, l \vDash e_l \Rightarrow n_1 \quad C, \sigma, l \vDash e'_l \Rightarrow n_2}{C, \sigma, l \vDash e_l - e'_l \Rightarrow n_1 - n_2}$$

$$\text{SV:L:Times}\quad \frac{C, \sigma, l \vDash e_l \Rightarrow n}{C, \sigma, l \vDash n' \times e_l \Rightarrow n' \times n}$$

Table 4 Constraint Syntax

$\mathcal{A}$	::=	$\mathbf{true} \mid u\ op\ u$		Constraint
	$\mid$	$\mathcal{A} \wedge \mathcal{A} \mid \mathcal{A} \vee \mathcal{A} \mid \mathcal{A} \Rightarrow \mathcal{A} \mid \neg\mathcal{A}$		
u	::=	$n \mid v \mid u + u \mid u - u \mid n \times u$		Expression
v	::=	$X \mid N(\ell_l) \mid N(l, \ell_l) \mid B(p, t_r, t_p)$		Variables
	$\mid$	$R(l, r) \mid O(l, o) \mid \|\mathcal{A}\|$		

packages) installed on the location l (for a package, this number is either 0 or 1);

- $B(p, t_r, t_p)$ is the number of bindings on the port p between the instances of the requiring type t_r and the providing type t_p;
- $R(l, r)$ is either 0 or 1, and expresses whether the repository r is installed on the location l;
- $O(l, o)$ tells how many of resource o the location l provides.

The semantics of our constraints is the same as usual: a solution σ for a constraint $\mathcal{A}$ is a mapping from the variables in $\mathcal{A}$ to integers, such that substituting the variables by their values in $\mathcal{A}$ will result in a tautology. For completeness, we present in Table 5 the semantics of a constraint, noting $\sigma \Vdash \mathcal{A}$

Table 5 Constraint Validation

$$\text{SV:True}\quad \sigma \Vdash \mathbf{true}$$

$$\text{CV:Exp}\quad \frac{\sigma \Vdash u \Rightarrow n \qquad \sigma \Vdash u' \Rightarrow n' \quad n\ op\ n'}{\sigma \Vdash u\ op\ u'}$$

$$\text{CV:And}\quad \frac{\sigma \Vdash \mathcal{A}_1 \qquad \sigma \Vdash \mathcal{A}_2}{\sigma \Vdash \mathcal{A}_1 \wedge \mathcal{A}_2}$$

$$\text{CV:Not}\quad \frac{\sigma \nVdash \mathcal{A}}{\sigma \Vdash \neg\mathcal{A}}$$

$$\text{CV:Or1}\quad \frac{\sigma \Vdash \mathcal{A}_1}{C, \sigma \Vdash \mathcal{A}_1 \vee \mathcal{A}_2}$$

$$\text{CV:Or2}\quad \frac{\sigma \Vdash \mathcal{A}_2}{\sigma \Vdash \mathcal{A}_1 \vee \mathcal{A}_2}$$

$$\text{CV:Imply1}\quad \frac{\sigma \Vdash \mathcal{A}_1 \qquad \sigma \Vdash \mathcal{A}_2}{\sigma \Vdash \mathcal{A}_1 \Rightarrow \mathcal{A}_2}$$

$$\text{CV:Imply2}\quad \frac{\sigma \nVdash \mathcal{A}_1}{\sigma \Vdash \mathcal{A}_1 \Rightarrow \mathcal{A}_2}$$

$$\text{CV:Plus}\quad \frac{\sigma \Vdash u_1 \Rightarrow n_1 \quad \sigma \Vdash u_2 \Rightarrow n_2}{\sigma \Vdash u_1 + u_2 \Rightarrow n_1 + n_2}$$

$$\text{CV:Minus}\quad \frac{\sigma \Vdash u_1 \Rightarrow n_1 \quad \sigma \Vdash u_2 \Rightarrow n_2}{\sigma \Vdash u_1 - u_2 \Rightarrow n_1 - n_2}$$

$$\text{CV:Times}\quad \frac{\sigma \Vdash u \Rightarrow n}{\sigma \Vdash n' \times u \Rightarrow n' \times n}$$

$$\text{CV:Number}\quad \sigma \Vdash n \Rightarrow n \qquad \text{CV:Var}\quad \sigma \Vdash v \Rightarrow \sigma(v)$$

$$\text{CV:Reify1}\quad \frac{\sigma \Vdash \mathcal{A}}{\sigma \Vdash \|\mathcal{A}\| \Rightarrow 1}$$

$$\text{CV:Reify2}\quad \frac{\sigma \nVdash \mathcal{A}}{\sigma \Vdash \|\mathcal{A}\| \Rightarrow 0}$$

when the mapping σ is a solution for $\mathcal{A}$. The external solver that we use, like FaCiLe [BB01] or Gecode [SLT] implement such semantics.

Another important semantics of these constraints in our case concerns configurations. Indeed, as we use these constraints to encode universes and specifications, we need to prove that our encoding is correct, i.e. the configurations validating a universe (resp. a specification) are exactly the same as the one validating their encoding. And so, we need the notion of a configuration validating a constraint. This notion is quite intuitive: to every configuration C corresponds a mapping σ from the special variables of our constraint syntax to the number of such elements in C ((for instance mapping $N(l,t)$ to $\#C(l,t)$). The configuration is a solution if its corresponding mapping is a solution. Formally, things are a littl bit more complicated, as a constraint can contain normal variables also. Our notion of validation for configuration is thus formalized in the following definition:

Definition 10. *Suppose given a constraint $\mathcal{A}$: we note $\mathcal{A}_l$ the set of location names used in $\mathcal{A}$. A configuration $C = \langle L, W, B\rangle$ and a universe $\mathcal{U}$* validates *$\mathcal{A}$ (noted $C,\mathcal{U} \vdash \mathcal{A}$) iff $C_l = \mathcal{A}_l$ and there exists σ with $\sigma \Vdash \mathcal{A}$ such that:*

$$\begin{cases}
\forall N(t) \in \mathcal{A},\ \sigma(N(t)) = \sum_{l \in C_l} \#(C(l,t)) & \forall N(l,t) \in \mathcal{A},\ \sigma(N(l,t)) = \#(C(l,t)) \\
\forall N(k) \in \mathcal{A},\ \sigma(N(k)) = \sum_{l \in C_l} \#(C(l,k)) & \forall N(l,k) \in \mathcal{A},\ \sigma(N(l,k)) = \#(C(l,k)) \\
\forall R(l,r) \in \mathcal{A},\ \sigma(R(l,r)) = 1 \Leftrightarrow C(l).r = r & \forall O(l,o) \in \mathcal{A},\ \sigma(O(l,o)) = C(l).\phi(o) \\
\forall N(p) \in \mathcal{A},\ \sigma(N(p)) = \sum_{l \in C_l, t \in \mathcal{U}^{\mathbf{P}}(p)} \#(C(l,t)) \times \mathcal{U}(t).\mathbf{P}(p) & \\
\forall N(l,p) \in \mathcal{A},\ \sigma(N(l,p)) = \sum_{t \in \mathcal{U}^{\mathbf{P}}(p)} \#(C(l,t)) \times \mathcal{U}(t).\mathbf{P}(p) & \\
\forall B(p,t_r,t_p) \in \mathcal{A},\ \sigma(B(p,t_r,t_p)) = \#(\{(p,c_r,c_p) \in B \mid C.\mathtt{type}(c_r) = t_r \wedge C.\mathtt{type}(c_p) = t_p\}) &
\end{cases}$$

In the rest of this section, we suppose fixed a universe $\mathcal{U}$, a specification S, an initial configuration C and an optimization function $\mathcal{F}$: the rest of this

Table 6 Universe Translation

$$\bigwedge_{p \in \mathcal{U}_{dp}} \begin{cases} \bigwedge\limits_{t_r \in \mathcal{U}_{\mathbf{R}}(p)} \mathcal{U}(t_r).\mathbf{R}(p) \times N(t_r) \leq \sum\limits_{t_p \in \mathcal{U}_{\mathbf{P}}(p)} B(p, t_p, t_r) \\ \bigwedge\limits_{t_p \in \mathcal{U}_{\mathbf{P}}(p)} \mathcal{U}(t_p).\mathbf{P}(p) \times N(t_p) \geq \sum\limits_{t_r \in \mathcal{U}_{\mathbf{R}}(p)} B(p, t_p, t_r) \\ \bigwedge\limits_{t_r \in \mathcal{U}_{\mathbf{R}}(p)} \bigwedge\limits_{t_p \in \mathcal{U}_{\mathbf{P}}(p)} B(p, t_p, t_r) \leq N(t_r) \times N(t_p) \\ \bigwedge\limits_{t \in \mathcal{U}_{\mathbf{C}}(p)} N(t) \geq 1 \Rightarrow N(p) = \mathcal{U}(t).\mathbf{P}(p) \end{cases} \tag{8}$$

$$\begin{cases} \bigwedge\limits_{t \in \mathcal{U}_{dt}} N(t) = \sum\limits_{l \in C_l} N(l, t) \wedge \bigwedge\limits_{k \in \mathcal{U}_{dk}} N(k) = \sum\limits_{l \in C_l} N(l, k) \\ \bigwedge\limits_{p \in \mathcal{U}_{dp}} N(p) = \sum\limits_{l \in C_l} N(l, p) \end{cases} \tag{9}$$

$$\bigwedge_{l \in C_l} \bigwedge_{p \in \mathcal{U}_{dp}} N(l, p) = \sum_{t_p \in \mathcal{U}_{\mathbf{P}}(p)} \mathcal{U}(t_p).\mathbf{P}(p) \times N(l, t_p) \tag{10}$$

$$\bigwedge_{l \in C_l} \begin{cases} \sum\limits_{r \in \mathcal{U}_{dr}} R(l, r) = 1 \\ \bigwedge\limits_{r \in \mathcal{U}_{dr}} R(l, r) = 1 \Rightarrow \begin{cases} \bigwedge\limits_{k \in \mathcal{U}(r)} N(l, k) \leq 1 \\ \bigwedge\limits_{k \in \mathcal{U}_{dk} \setminus \mathcal{U}(r)} N(l, k) = 0 \end{cases} \end{cases} \tag{11}$$

$$\bigwedge_{l \in C_l} \begin{cases} \bigwedge\limits_{t \in \mathcal{U}_{dt}} N(l, t) \geq 1 \Rightarrow \sum\limits_{k \in \mathcal{U}_i(t)} N(l, k) \geq 1 \\ \bigwedge\limits_{k_1 \in \mathcal{U}_{dk}} \bigwedge\limits_{K \in \mathcal{U}(k_1).\mathbf{R}} N(l, k_1) \leq \sum\limits_{k_2 \in K} N(l, k_2) \\ \bigwedge\limits_{k_1 \in \mathcal{U}_{dk}} \bigwedge\limits_{k_2 \in \mathcal{U}(k_1).\mathbf{C}} N(l, k_1) + N(l, k_2) \leq 1 \end{cases} \tag{12}$$

$$\bigwedge_{l \in C_l} \bigwedge_{o \in \mathcal{O}} \sum_{x \in \mathcal{U}_{dt} \cup \mathcal{U}_{dk}} \mathcal{U}(x).\mathbf{f}(o) \times N(x, l) \leq O(l, o) \tag{13}$$

$$\begin{cases} \bigwedge\limits_{t \in C_t \setminus \mathcal{U}_{dt}} N(t) = 0 \\ \bigwedge\limits_{k \in C_k \setminus \mathcal{U}_{dk}} N(k) = 0 \end{cases} \tag{14}$$

section presents how we translate these data into a constraint. Note that the set of locations C_l (coming from the input initial configuration C) is used in all aspects of our translation, as it corresponds to the set of locations on which the different elements of the configuration (components, packages) will be installed.

4.2 Universe Translation

We present in Table 6 our translation of the universe $\mathcal{U}$ into a constraint $\mathcal{A}$. Equation 8 encodes the dependencies between component types: the first line states that all the requirements of a type must be satisfied by some bindings; the second line states that providers cannot have more output bindings than

what they provide; the third line states that there shouldn't be two bindings on the same port between the same components;[5] and the fourth line ensures that when a type is in conflict with a port, there is no other component providing that port in the configuration. Equation 9 encodes the distribution of components, ports and packages: for every element ℓ_l, the number of instances of ℓ_l in the configuration is the sum of its instances on each location. Equation 10 states that the number of a ports in a location is equal to how many times that port is provided. Equation 11 encodes repository installation: the first line states that exactly one repository must be installed in a location; and the second line states when the repository r is installed on a location l, only the packages of that repository can be installed on l. Equation 12 encodes the three relations involving packages: the first line encodes the implementation relation between component types and packages; the second line encodes the dependency relation between packages; and the third line encodes the conflicts between packages. Equation 13 encodes resource usage in each location. Finally, equation 14 ensures that all components and packages unvalid in the universe are removed from the initial configuration. Our encoding enjoys the following property:

Lemma 1. *Given the constraint $\mathcal{A}$ generated from $\mathcal{U}$ and a configuration C', then C',$\mathcal{U}$ validates $\mathcal{A}$ iff C' validates $\mathcal{U}$.*

Sketch. Let consider the three parts of the definition of $\mathcal{U} \vdash C'$. We can quite easily see that the equations 8, 9 — instantiated for component types —, 10, the first line of 12 and the first line of 14 are equivalent to C' being component-valid for $\mathcal{U}$. We can quite easily see that the equations 11, the two last lines of 12 and the second line of 14 are equivalent to C' being package-valid for $\mathcal{U}$. We can quite easily see that the equation 13 is equivalent to C' being resource-valid for $\mathcal{U}$. □

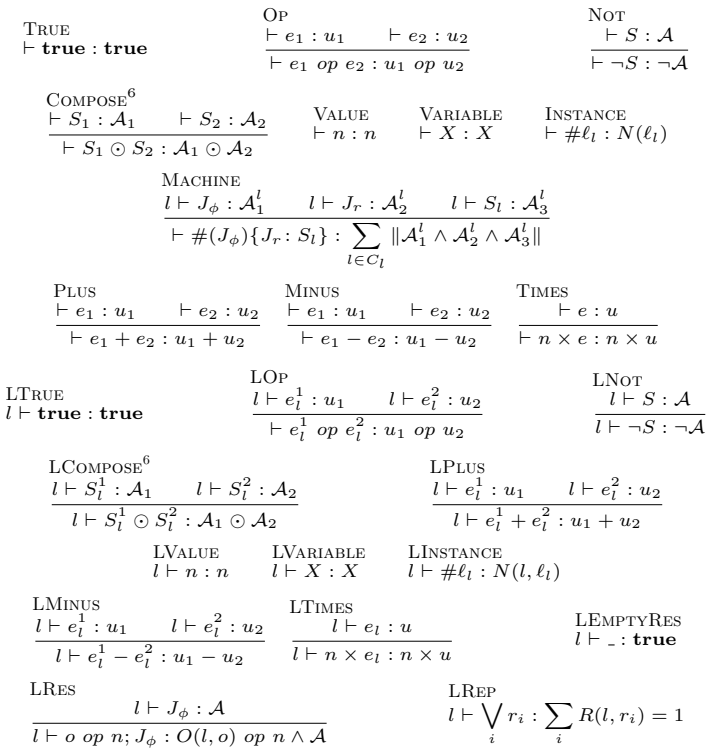
Basically, this lemma states that any configuration that validates the generated constraint is correct w.r.t. the input universe.

4.3 Specification Translation

We present in Figure 7 our translation of a specification S into a constraint $\mathcal{A}$. Our translation is done by induction on the structure of S, and uses statements of the forms $\vdash S : \mathcal{A}$ for specifications, $\vdash e : u$ for expressions, $l \vdash S_l : \mathcal{A}$ for expressions local to the location l, and $l \vdash e_l : u$, $l \vdash J_\phi : \mathcal{A}$ and $l \vdash J_r : \mathcal{A}$ for expressions, resource and repository constraints local to l. The resulting constraint is almost identical to S: only the references to elements ℓ, resources and repository constraints have been translated into their equivalent in the constraint syntax. The most interesting rules in our translation are INSTANCE,

[5] One can remark that this constraint is not linear, and non-linear constraints solving is in general undecidable. Fortunately, this particular constraint can be translated into several linear ones.

Table 7 Specification Translation



MACHINE, LINSTANCE, LRES and LREP. Rule INSTANCE states that $\#\ell_l$ which corresponds to the number of instances of ℓ_l in the configuration, is the variables $N(\ell_l)$. Rule MACHINE counts the number of locations validating the specification in input: to do so, it takes all the locations l in the configuration, checks if that location validates the specification, and if yes, adds one to the count, using reified constraints. Rule LINSTANCE applies when $\#\ell_l$ is used inside the specification of a location: in that case, $\#\ell_l$ corresponds to the number of instances of ℓ_l in that location, and thus, is the variable $N(l, \ell_l)$. Rule LRES encodes constraints on resources available on a location using the variables $O(l, o)$. Finally, rule LREP encodes the fact that only the repositories r_i can be installed on l with a sum ensuring that one of the $R(l, r_i)$ is equal to one.

Our encoding enjoys the following property:

Lemma 2. *Given the constraint $\mathcal{A}$ generated from S, a configuration C and a universe $\mathcal{U}$, then C,$\mathcal{U}$ validates $\mathcal{A}$ iff they validates S.*

Sketch. As the translation process is almost a one to one correspondence, the result is direct for most cases, except for the MACHINE rule. That rule encodes

the number of locations validating an inner constraint into a sum of numbers being 0 if a location does not validate the constraint, and 1 otherwise. That sum is thus eaxactly equals to the number of locations validating the inner constraint, which gives us the result. □

4.4 Initial Configuration Translation

From the initial configuration, we extract the informations concerning the available locations. This was already partially done in the previous constraints, where we used variables of the form $N(l, \ell_l)$. It remains to encode as constraint the resources that are available on these locations. This is done with the following constraint:

$$\bigwedge_{l \in C_l} \bigwedge_{o \in \mathcal{O}} O(l, o) = C(l).\phi(o) \tag{15}$$

Here, we simply give the value of all the variables $O(l, o)$.

4.5 Optimization Function

Currently we support 3 different optimization functions:

1. `compact` aims to use the least number of location possible. This corresponds to the following formula:

$$\min \left(\sum_{l \in C_l} \| \sum_{k \in \mathcal{U}_{dk}} N(l, k) \geq 1 \| \right)$$

   The sum counts all the locations that are used (i.e. all the locations on which a package is installed). The goal of the optimization is then to minimize that number.

2. `spread` aims to use the least number of components and packages, and to place them on a maximal number of locations, to fully use the available resources of the configuration. We built a function with that semantics using a lexicographic order, that first minimizes the number of components and packages in the system, and then maximizes the number of used locations. This results in the following formula:

$$\text{lex} \left(\min(\sum_{x \in \mathcal{U}_{dt} \cup \mathcal{U}_{dk}} N(x)) \; ; \max(\sum_{l \in C_l} \| \sum_{k \in \mathcal{U}_{dk}} N(l, k) \geq 1 \|) \right)$$

3. `conservative` finally aims to get a configuration that is the closest to the initial one. To do that, our optimization function minimizes the difference

[6] For concision, $\odot$ stands for either $\wedge$, $\vee$ or $\Rightarrow$

between the two configurations. Namely, it minimizes the difference in which packages and components are installed on each locations:

$$\min \left(\sum_{l \in C_l} \left\{ \begin{array}{l} \sum\limits_{t \in \mathcal{U}_{dt}} |N(l,t) - \#C(l,t)| \\ \sum\limits_{k \in \mathcal{U}_{dk}} |\, N(l,k) - \|C(l,k)\| \,| \end{array} \right. \right)$$

5 Configuration Generation

We now suppose that an optimal solution for the constraints has been found by the solver. In this section, we present how Zephyrus generates its output configuration $C' = \langle L', W', B' \rangle$ from that solution and the initial configuration $C = \langle L, W, B \rangle$.

5.1 Location Generation

First, Zephyrus generates the set of locations L'. This is simply done by taking the locations from the initial configuration, and configuring them as described in the solution (i.e. installing the right repositories and packages). We choose not to remove the unused locations from the configuration, to leave that choice to the system designer. Formally, L' is defined as follow:

- $dom(L') = dom(L)$: the set of locations is the same the initial configuration;
- $\forall l \in C_l, L'(l) = \langle L(l).\phi, r, \{k \,|\, N(l,k) = 1\} \rangle$ where $R(l,r) = 1$: for each location, the resource it provides is the same as before, while the repository and the packages it hosts are defined by the solution of the constraint.

5.2 Component Generation

Second, Zephyrus generates the components running on the system. To make the runtime redeployment of the system as efficient as possible (and to comply with the `conservative` optimization function), we try to reuse as many existing components as possible. To achieve this, we use the sets $J_{l,t}$ and $I_{l,t}$ that respectively correspond to the components on location l with type t that we *reuse* from the initial configuration, and the ones that we *generate* to get the resulting configuration. These sets are defined as follow:

- $J_{l,t}$ is one of the biggest subset of $C(l,t)$ whose cardinality is smaller than $N(l,t)$. This means that if there are too many components of type t on l in the initial configuration then we remove some of them to get only $N(l,t)$ of them; and if there are less components than $N(l,t)$ then we keep all of them, and add new ones with the set $I_{l,t}$. Formally, $J_{l,t}$ is defined as follows:

$$\forall l \in C_l, t \in \mathcal{U}_{dt} \left\{ \begin{array}{l} J_{l,t} \subset C(l,t) \\ \#(J_{l,t}) = \min(\#(C(l,t), N(l,t)) \end{array} \right.$$

- $I_{l,t}$ is the set of components of type t on location l that we add to the initial configuration to fit the cardinality $N(l,t)$ found by the constraint solving:

$$\forall l \in C_l, t \in \mathcal{U}_{dt} \begin{cases} I_{l,t} \text{ fresh} \\ \#(J_{l,t} \cup I_{l,t}) = N(l,t) \end{cases}$$

Using these sets, the construction of W' is quite direct: the components in C' are the $J_{l,t}$ and $I_{l,t}$, and as we described, all components in $J_{l,t}$ or $I_{l,t}$ are in location l, with the type t. Formally, we have

$$\begin{cases} dom(W') = \bigcup_{l \in C_l} \bigcup_{t \in \mathcal{U}_{dt}} (J_{l,t} \cup I_{l,t}) \\ \forall l \in C'_l, t \in \mathcal{U}_{dt}, c \in J_{l,t} \cup I_{l,t}, W'(c) = \langle l, t \rangle \end{cases}$$

5.3 Binding Generation

This last step of the construction of the configuration is the most difficult one. As presented in the last lines of Table 8, the principle is quite simple: for each component, we look at its dependencies, find a set of providers to satisfy them, and then construct the binding accordingly. The difficult part is the choice of these providers, which must follow three constraints: i) we must respect the number given by the solution of the generated constraint; ii) we cannot bind a provider too many times; and iii) all the bindings must be unique. This part is done in the `select` function which is based on two tables: Tp gives for all ports p the set of components c providing p, together with how many client that component can still be connected to; Tt gives the number of bindings still to be created between the instances of each component type. Basically, Tp is used to ensure that we respect the connection capacity of each provider, and Tt is used to ensure that we follow the solution of the constraint. On the other hand, the unicity of the bindings, as well as the completeness of the algorithm is ensured by the **for** loop in the `select` function. This **for** loop has three main features. First, it takes a pair (n, c) at most once from Tp, ensuring the unicity of the generated bindings; Second, it takes that pair in *decreasing order*, i.e. it first takes the providers with the highest capacity. The idea is to keep as many available providers as possible (i.e. with $n > 0$) to ensure the completeness of the algorithm. Finally, the loop is finished by an **until** statement that ensures we pick the right number of providers.

Lemma 3. *Given a solution σ to the constraint generated in Section 4, the binding generation algorithm will create as many bindings as specified by the different values $\sigma(B(p, t_r, t_p))$.*

Sketch. In addition to the two tables Tt and Tp used in the algorithm, consider the table Tr mapping all pairs (p, t) where p is a port and $t \in \mathcal{U}\mathtt{R}(p)$ to the number of component of type t for which the bindings on p aren't defined yet. We also note $Tp^+(p, t)$ the subset of $Tp(p)$ where the components are of type t

[7] For convenience, we note $c.C_t$ the type of the component c in the configuration C'.

Table 8 Binding Generation Algorithm[7]

// Selection algorithm
$\forall p \in \mathcal{U}_{dp},\ Tp(p) \leftarrow \{(c,n) \mid c.C_t \in \mathcal{U}\mathtt{P}(p) \wedge n = \mathcal{U}(c.C_t).\mathbf{P}(p)\}$
$\forall p \in \mathcal{U}_{dp}, t_r \in \mathcal{U}\mathtt{R}(p), t_p \in \mathcal{U}\mathtt{P}(p),\ Tt(p,t_r,t_p) \leftarrow B(p,t_r,t_p)$

```
select(p, t_r) {
  res ← ∅
  for (c,n) ∈ Tp(p) in decreasing order {
    if Tt(p,t_r,c.C_t) ≠ 0 {
      res ← res ∪ {c}
      Tt(p,t_r,c.C_t) --
    }
  } until (#(res) = U(t_r).R(p))
  for c ∈ res { replace (c,n) with (c,n−1) in Tp(p) }
  return res
}

// Main algorithm
B' ← ∅
for c ∈ W' {
  for p ∈ dom(U(c.C_t).R) {
    G ← select(p, c.C_t)
    B' ← B' ∪ {(p,c,c') | c' ∈ G}
}}
```

and are mapped to strictly positive integers. It is not difficult to see that the inner loop of the main part of the algorithm enjoys the three following invariants (derived from (8)):

$$\bigwedge_{p \in \mathcal{U}_{dp}} \begin{cases} \bigwedge\limits_{t_r \in \mathcal{U}\mathtt{R}(p)} (\mathcal{U}(t_r).\mathbf{R}(p) \times Tr(p,t_r) = \sum\limits_{t_p \in \mathcal{U}\mathtt{P}(p)} Tt(p,t_p,t_r)) \\ \sum\limits_{t_p \in \mathcal{U}\mathtt{P}(p)} n \geq \sum\limits_{t_r \in \mathcal{U}\mathtt{R}(p)} Tt(p,t_p,t_r)) \\ (c,n) \in Tp^+(p,t_p) \\ \bigwedge\limits_{t_r \in \mathcal{U}\mathtt{R}(p)} \bigwedge\limits_{t_p \in \mathcal{U}\mathtt{P}(p)} Tt(p,t_p,t_r) \leq Tr(p,t_r) \times \#Tp^+(p,t_p) \end{cases}$$

Now, consider a component c of type t_r, and a port p such that $p \in dom(\mathcal{U}(t_r).\mathbf{R})$: we show that the `select` function will find enough providers to satisfy the requirements of c, thus proving correctness and completeness of our algorithm. First, as c must be bound, the first invariant tells us that

$$\mathcal{U}(t_r).\mathbf{R}(p) \leq \sum_{t_p \in \mathcal{U}\mathtt{P}(p)} Tt(p,t_p,t_r)) \quad \text{which implies} \quad \mathcal{U}(t_r).\mathbf{R}(p) \leq \sum_{t_p \in \mathcal{U}\mathtt{P}(p)} \#Tp^+(p,t_p)$$

This means that we have enough available providers to satisfy c, and by construction of its **for** loop which takes providers in decreasing order, the `select` function will find them. □

5.4 Properties

We finally generate all the elements L', W' and B' defining the output configuration C' of Zephyrus. We can now state the main properties of C': soundness, completeness and optimality.

Theorem 1 (Soundness)**.** *The computed configuration C' validates the input universe $\mathcal{U}$, specification S and uses the locations given in the input configuration C.*

Sketch. The fact that C' validates the generated constraint is direct from how we constructed it, except for the bindings whose generation algorithm is proved correct and complete in the appendix. This implies, by Lemma 1 and 2, that C' indeed validates the input universe $\mathcal{U}$ and specification S. Finally, by construction, C' uses the locations given in the input configuration C. □

Theorem 2 (Completeness)**.** *If there exists a configuration C'' that validates the input universe $\mathcal{U}$, specification S and is deployed on the locations of C, then Zephyrus will succesfully compute some solution C'.*

Sketch. By Lemma 1 and 2, we can see that C'' validates the constraint $\mathcal{A}$ generated in Section 4. Hence, $\mathcal{A}$ has a solution, which means that the solver succeeds to produce a solution. And finally, our configuration generation algorithm, which never fails, will produce C'. □

Theorem 3 (Optimality)**.** *The generated configuration C' is optimal w.r.t the chosen optimization function.*

Sketch. By definition, the solution given by the solver is optimal w.r.t. the optimization function. By construction, C' follows that solution in its design, and thus, is optimal too. □

6 Related and Future work

The problem of managing networks of interconnected machines has attracted significant attention in the area of system administration. Many popular tools to that end exist, e.g. CFEngine [Bur95], Puppet [Kan06], MCollective [Pup] and Chef [Ops]. Despite their differences, they all allow to declare the components to be installed on each machine, together with their configuration files, and then employ various mechanisms to deploy components accordingly. The burden of specifying which components to deploy where, and how to interconnect them is left to the user, let alone the difficult problem of optimal resource allocation. As an additional complication, these tools rely blindly on existing package managers, and they have no way of knowing in advance whether package installation will actually succeed: if the user requests to install two web servers on the same machine, the incompatibility will only be discovered at deploy time, when one of the services fails to get installed (or start). In our approach incompatibilities are known to Zephyrus that can then plan around them. System

management tools can however be used as convenient deployment backend for Zephyrus: once optimal resource allocation is done, the actual deployment can be delegated to them, with the guarantee that no deployment error will arise. CloudFoundry [VMW], while specifically targeted at application deployment in the cloud, has the same limitations described above.

ConfSolve [HAG12] improves on the tools described above, relying on a constraint solver to propose an optimal allocation of virtual machines on servers and applications on virtual machines, but it does not handle in any way neither connections among services, nor capacity or replication constraints, and knows nothing about package incompatibilities.

Two recent efforts, Juju [Can] and Engage [FME12], are more similar to our approach: they both rely on a solver to perform their deployment work. But there are several major differences with our work. First, both tools avoid the problem of dealing with conflicts among components. In Juju, each service is deployed on a single machine (or, more recently, in a virtual container inside one). That avoids conflict issues, but at the price of wasting resources: in the example of Section 2 Zephyrus proposes a solution that needs 4 machines, whereas Juju would have required 6 (or 7, in the increased redundancy case). In Engage, conflicts are not even available in the specification language, one can only indicate that a service can be realised by exactly one out of a list of components. Second, neither of these tools—or any other that we are aware of—allows to declare capacity or replication constraints, which are essential in any non-trivial, scalable application. Finally, none of the aforementioned tools allows to find a deployment that uses resources in an optimal way, minimizing the number of needed (virtual) machines.

Another approach to automating deployment is proposed in [ECBdP11]; it uses an Architecture Description Language with information on the relationships among software services, which needs to be explicitly provided by the user in full detail, and uses a decentralized protocol to perform automatic configuration. This work may also be used as a backend for Zephyrus.

In future work, we plan to study under which assumptions one can also produce a detailed reconfiguration plan for stopping and restarting the deployed services in the right order to minimize downtime. We also plan to extend the current model, which is "flat" in the sense of [DCZZ12], to support hierarchies of deployment locations to represent both administrative domains (such as connected private networks) and nested virtualization containers. Something similar has been done in Engage, but without any support for restricting the visibility of components according to placement in the hierarchy. We would like to support that, considering visibility to be the most useful feature of hierarchies, especially in the presence of conflicts.

7 Conclusion

We have described a concise and powerful, semi-automated approach to the design and deployment of complex distributed applications composed of inter-

connected services, as they are typically found in modern cloud environments. The system architect can specify the core components needed to obtain the required functionalities, add non functional constraints—like a maximum number of clients connected to a given service, or a minimum number of replicas—as well as constraints on physical resources—e.g. memory or bandwidth—and explicit incompatibilities among components. The user can also choose among various optimization functions, which allow to specify whether she prefers a conservative solution, changing the current configuration as little as possible, or a highly economical solution, using only a minimum number of machines.

Equipped with all this, the prototype tool Zephyrus will find an optimal deployment solution and output a complete system configuration, including precise information about service interconnection. Such a description can then be used as input for traditional low-level configuration management systems which are popular in system administration circles. 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 shielding from deploy-time errors. We have also formally proved that our encoding is correct w.r.t the specification, and that finding the correct service interconnections will always succeed when a solution is found.

To the best of our knowledge, this is the first tool that allows to handle capacity and replication constraints, conflicts, and multiple services on a single machine, thus finally providing an instrument able to handle the stringent requirements of distributed applications in the real world.

References

[BB01] Pascal Brisset and Nicolas Barnier. Facile: a functional constraint library, 2001.

[Bur95] Mark Burgess. A site configuration engine. *Computing Systems*, 8(2):309–337, 1995.

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

[CV11] Roberto Di Cosmo and Jérôme Vouillon. On software component co-installability. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT FSE*, pages 256–266. ACM, 2011.

[DCZZ12] Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro. Towards a formal component model for the cloud. In *SEFM 2012*, volume 7504 of *LNCS*, pages 156–171. Springer, 2012.

[ECBdP11] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-configuration of distributed applications in the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 668 –675, july 2011.

[FME12] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. In *PLDI'12: Programming Language Design and Implementation*, pages 263–274. ACM, 2012.

[HAG12] John A. Hewson, Paul Anderson, and Andrew D. Gordon. A declarative approach to automated configuration. In *LISA '12: Large Installation System Administration Conference*, pages 51–66, 2012.

[Kan06] Luke Kanies. Puppet: Next-generation configuration management. *`;login:` the USENIX magazine*, 31(1):19–25, 2006.

[MBC+06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE*, pages 199–208. IEEE Computer Society, 2006.

[ND11] 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.

[Ops] Opscode. Chef. `http://www.opscode.com/chef/`. Retrieved February 2013.

[Pup] Puppet Labs. Marionette collective. http://docs.puppetlabs.com/mcollective/. Retrieved February 2013.

[SLT] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode. http://www.gecode.org/. Retrieved February 2013.

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