Zephyrus2: On the Fly Deployment Optimization using SMT and CP Technologies[⋆] - Technical Report

Erika Ábrahám[1], Florian Corzilius[1], Einar Broch Johnsen[2], Gereon Kremer[1], and Jacopo Mauro[2]

[1] RWTH Aachen University, Germany
[2] Department of Informatics, University of Oslo, Norway

Abstract. Modern cloud applications consist of software components deployed on multiple virtual machines. Deploying such applications is error prone and requires detailed system expertise. The deployment optimization problem is about how to configure and deploy applications correctly while at the same time minimizing resource cost on the cloud. This problem is addressed by tools such as Zephyrus, which take a declarative specification of the components and their configuration requirements as input and propose an optimal deployment. This paper presents Zephyrus2, a new tool which addresses deployment optimization by exploiting modern SMT and CP technologies to handle larger and more complex deployment scenarios. Compared to Zephyrus, Zephyrus2 can solve problems involving hundreds of components to be deployed on hundreds of virtual machines *in a matter of seconds instead of minutes.* This significant speed-up, combined with an improved specification format, enables Zephyrus2 to interactively support on the fly decision making.

1 Introduction

Modern software systems are often developed to be highly configurable both in the functionality they offer and in their deployment architecture. Applications targeting the cloud need to adapt their deployment to the virtual machines (VMs) that the cloud makes available. Cloud applications typically consist of a large number of interconnected software components (such as packages or services) that must be deployed on VMs that can be created on-the-fly by means of cloud computing technologies. The correct deployment and configuration of cloud applications is a challenging task and a major source of errors. In fact, inappropriate deployment and configuration are the second cause of errors in Google data centers, only after software bugs [5]. The deployment flexibility of cloud applications is further restricted by the availability and price of resources offered by the cloud. The *deployment optimization problem* is the problem of how to correctly deploy all the software components needed by a cloud application on suitable VMs on the cloud at minimal cost.

The deployment and on-the-fly configuration of systems on the cloud are handled by so-called DevOps teams, which address efficient system delivery and frequent

[⋆] Supported by the EU project FP7-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems* and by the DAAD project SMT4ABS.

infrastructure changes by combining development and operations experts. Different tools and technologies have been developed to support the work of DevOps teams. The mainstream approach restricts solutions to a fixed set of pre-configured VM images, which offers all the needed software packages and services, and which can be launched directly on the VMs of the targeted cloud system (e.g., Bento Boxes [23], Cloud Blueprints [11], or AWS CloudFormation [3]). The main drawback of this approach is that, as the deployment may use only the pre-configured VM images, it might use more resources than necessary or force the software to only run on specific cloud providers (resulting in vendor lock-in). More advanced techniques allow application architects to design their own software architectures using *high-level description languages* such as the graphical drag-and-drop approach of Juju [30] or the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard [40]. Unfortunately, the use of these languages is knowledge-intensive since they require the architect to design the entire architecture and have a deep understanding of all the components to deploy. Furthermore, these languages do not address deployment optimization.

To overcome these limitations and address the deployment optimization problem, *declarative approaches* have recently been proposed which enable the DevOps teams to automatically generate optimal VM configurations from high-level specifications [22, 29]. In particular, the automatic configuration generator tool *Zephyrus* [16] has been applied in a number of industrial settings [14, 18, 25]. Starting from a description of the available VMs and the components that need to be deployed, the architect can specify requirements in the form of constraints and use the tool to generate optimal machine configurations and deployment at minimal cost. The application architect can exploit the expressiveness of the constraints to focus on the most important aspects of the application, leaving to the tool the task of deducing other components that are needed to obtain a correct configuration and where to deploy them.

The contribution of this paper is to present *Zephyrus2*, a new tool to tackle the deployment optimization problem, inspired by Zephyrus. Zephyrus2 overcomes some limitations of Zephyrus by using different solving approaches, which enables us to solve problems involving hundreds of components to be deployed on hundreds of VMs *in a matter of seconds instead of minutes.* We report on the obtained performance gains with different solving approaches. Based on industrial experiences with declarative deployment optimization [14, 18, 25], Zephyrus2 allows a more direct and concise specification of deployment scenarios and user requirements than Zephyrus. The simplified input format combined with a significant speed-up (i.e., seconds instead of minutes) allows Zephyrus2 to be used by DevOps teams in a more interactive way for on the fly decision making [19].

Paper Structure. Section 2 gives a brief overview of the declarative deployment optimization problem. Section 3 introduces Zephyrus2 and Section 4 evaluates its performance on a set of industry inspired instances. Section 5 discusses the main differences between Zephyrus2 and Zephyrus and Section 6 discusses other related work. Section 7 concludes the paper, indicating directions for future research.

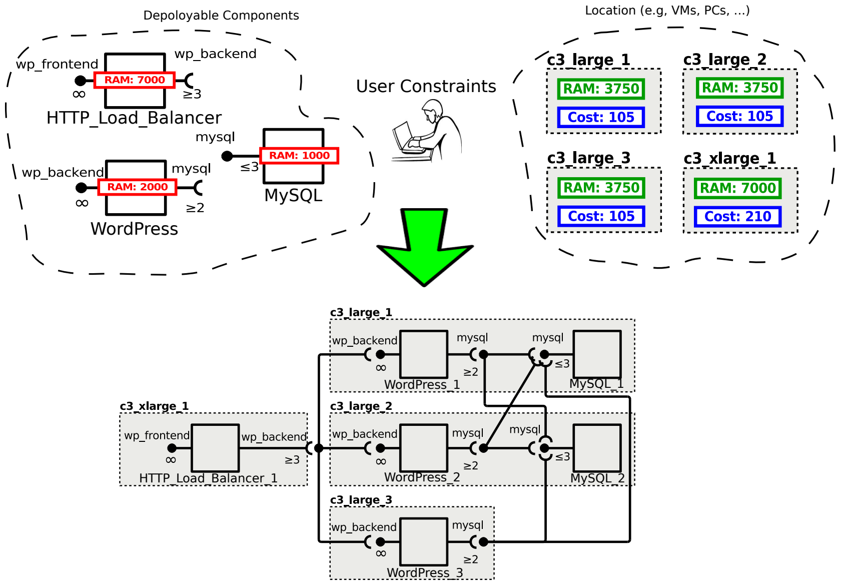


Fig. 1: The deployment problem.

2 Preliminaries

We give an overview of the declarative deployment optimization problem [10, 16]. The basic deployment problem is illustrated by Figure 1; we assume three different inputs:

(i) a description of the *components* that can be deployed,
(ii) a description of the *virtual machines* where the components can run, and
(iii) the *constraints* that capture the specific requirements of the DevOps teams.

We specify *components* in the Aeolus [17] modeling language as black-boxes that expose *require-* and *provide-ports* to capture required and provided functionalities respectively. *Connections* (*bindings*) from require- to provide-ports model the usage of services. *Capacity constraints* associated to the ports might constrain those connections: i) for provide-ports they can specify how many require-ports might be connected (maximal number of served components), or impose that no other component can provide the same functionality (used to model the notion of conflicts among components), ii) for require-ports multiple providers offering the given functionality can be required (used to model replication requirements). Every component instance consumes resources such as memory or processing power when deployed, which is also captured in the model. Some examples for component descriptions are graphically illustrated in Figure 1 (top left). For instance, the open-source content management system WordPress is represented by a component named `WordPress` which provides, if installed, the functionality `wp_backend` via a provide-port and which requires the functionality `mysql` via a require-port. By associating the ∞ symbol to the provide-port `wp_backend` we model that the functionality can be provided to an unbounded number of other components.

The `WordPress` component requires to be connected to at least two different components providing the `mysql` functionality (e.g., to provide fault tolerance). This is expressed by associating the capacity constraint "≥ 2" to the `mysql` require-port. In our example only the resource `RAM` is modeled: a `WordPress` instance is associated with the consumption of 2000 MB of RAM.

The *virtual machines* are modeled as *locations*. Each location has a name, a list of resources that it can provide, and an associated cost. Figure 1 (top right) shows four locations named `c3_large_1`, `c3_large_2`, `c3_large_3`, and `c3_xlarge_1`. They represent four different machines inspired by Amazon EC2, three of them are `c3_large`, providing 3.75 GB of RAM, and one of them is `c3_xlarge`, providing 7 GB of RAM. The instances of type `c3_large` have an associated cost of 105 to indicate that their cost is 0.105 dollars per hour.

The user can specify (*deployment*) *constraints* in an ad-hoc declarative language powerful enough to express, e.g., the presence of a given number of components and their co-installation requirements or conflicts. For the example in Figure 1, the user might require the presence of at least one `HTTP_Load_Balancer` and impose that, for fault tolerance reasons, no two `WordPress` or `MySQL` instances should be installed on the same virtual machine.

The goal of the *declarative deployment optimization problem* is to find a configuration distributing components on a set of locations such that:

(i) the constraints reflecting the user requirements are satisfied,
(ii) every functionality required by a deployed component is provided,
(iii) in each location, the available resources are sufficient to cover the resource needs of all components deployed on it, and
(iv) the values of some user-defined (prioritized) objective functions are minimized.

3 Zephyrus2

Zephyrus2 is a tool to solve the declarative deployment optimization problem. It offers a concise language to specify deployment optimization problems, and it can use different technologies to solve them. Zephyrus2 is written in Python ($\sim 3k$ lines of code) and is open source and freely available [33]. In order to increase portability, Zephyrus2 can be installed using Docker containers [21].

3.1 Problem Specification Language

The use of Zephyrus in an industrial environment [14, 18, 25] has emphasized the need to have (i) a simpler way to define components and locations, and (ii) a more concise specification language to describe deployment constraints.

To tackle the first concern, Zephyrus2 supports for component and location specifications the JavaScript Object Notation (JSON) format. Due to the lack of space, here we only show some examples; the formal JSON Schema of the input is available at [33]. The following JSON snippet defines the WordPress component in Figure 1:

```
"WordPress": {
  "resources": { "RAM": 2000 },
  "requires": { "mysql": 2 },
  "provides": [ { "ports": [ "wp_backend" ], "num": -1 } ]
}
```

In the second line, with the keyword `resources`, it is declared that WordPress consumes 2000 MB of RAM. The keyword `requires` defines that the component has a require-port requiring the service `mysql` with a capacity constraint "$\geq$ 2". Similarly, the `provides` keyword declares that WordPress provides `wp_backend` to a possibly unbounded number of components (represented by -1).

The definition of locations is also done in JSON. For instance, the JSON input to define 10 `c3_large` Amazon virtual machines is the following:

```
"c3_large": {
  "num": 10,
  "resources": { "RAM": 3750 },
  "cost": 105
}
```

To tackle the second concern, Zephyrus2 introduces a new specification language for deployment constraints. This language is a key factor for the usability of the tool: while users who want to deploy their applications on a cloud usually need rather simple deployment constraints (requiring, e.g., that one instance of the main application component should be deployed), the language allows DevOps teams to express also more complex cloud- and application-specific constraints. In the following we describe some main features of the language by means of simple examples, referring the interested reader to [33] for the formal grammar of the language and more examples.

A deployment constraint is a logical combination of comparisons between arithmetic expressions. Besides integers, expressions may refer to component names representing the total number of deployed instances of a component. Location instances are identified by a location name followed by the instance index (starting at zero) in square brackets. A component name prefixed by a location instance stays for the number of component instances deployed on the given location instance. For example, the following formula requires the presence of at least one HTTP Load Balancer instance, and exactly one WordPress server instance on the second `c3_large` location instance:

```
HTTP_Load_Balancer > 0 and c3_large[1].WordPress = 1
```

For quantification and for building sum expressions, we use identifiers prefixed with a question mark as variables. Quantification and sum building can range over components, locations, or over components/locations whose names match a given regular expression. Using such constraints, it is possible to express more elaborate properties such as the co-location or distribution of components, or limit the amount of components deployed on a given location. For example, the constraint

```
forall ?x in locations: ( ?x.WordPress > 0  impl ?x.MySQL > 0)
```

states that the presence of an instance of `WordPress` deployed on any location `x` implies the presence of an instance of `MySQL` deployed on the same location `x`. As another example, requiring the HTTP Load Balancer to be installed alone on a virtual machine can be done by requiring that if a Load Balancer is installed on a given location then the sum of the components installed on that location should be exactly 1.

```
forall ?x in locations: ( ?x.HTTP_Load_Balancer > 0 impl
  (sum ?y in components: ?x.?y) = 1 )
```

For optimization, Zephyrus2 allows the user to express her preferences over valid configurations in the form of a list of arithmetic expressions whose values should be minimized in the given priority order. The keyword `cost` can be used to require the minimization of the total cost of the application. The following list specifies the metric to minimize first the total cost of the application and then the total number of components:

```
cost; ( sum ?x in components: ?x )
```

This is also the default metric used if the user does not specify her own preferences.

3.2 Solving Technologies

Zephyrus2 solves deployment optimization problems, specified in the above-described languages, by translating them into *Constraint Optimization Problems* (*COP*) encoded in MiniZinc [38].[3] By default, Zephyrus2 solves the resulting multi-objective optimization problems by optimizing the first objective function value and then optimizing the other objective function values sequentially following their order after substituting the previously determined optimal values. We believe that this solution is particularly effective since usually minimizing the first objective (e.g., the cost) has a significant impact on the performance when reducing the second objective (e.g., the number of components). However, this solution has the drawback that we need to restart the solver.

In order to exploit the capabilities of further multi-objective optimization techniques, Zephyrus2 also supports MiniSearch [45], a meta-search language for MiniZinc that allows to solve MiniZinc models with (heuristic) meta-searches, such as large neighborhood search (LNS), lexicographic branch-and-bound, and And/Or search. Currently, Zephyrus2 uses MiniSearch to execute a lexicographic branch-and-bound search procedure. Unfortunately, for the time being, there is a limited amount of solvers supporting the programmatic APIs of the version 2.0 of MiniZinc that eliminates the need to communicate through text files and enables the addition of constraints at runtime without restarting the solvers. However, from an engineering point of view, we believe that the support of MiniSearch is important since it allows to explore and try different search procedures and improve the current performance as soon as more constraint programming solvers will adopt the MiniZinc 2.0 APIs.

Zephyrus2 also supports the use of *satisfiability-modulo-theories* (*SMT*) *solvers*. SMT solving extends and improves upon SAT solving by introducing the possibility of stating constraints in some expressive theories, e.g., arithmetic or bit-vector expressions. For our application, we need a solver that supports integer arithmetic and also features optimization. One that is capable of doing that is Z3 [6, 20], one of the most prominent SMT solvers. The last version of Z3 (4.4.2) has introduced some optimization features as an extension of the SMT-LIBv2 input language [4], i.e., the standard

[3] The MiniZinc encoding used by Zephyrus2 is reported in Appendix B.

format to define SMT instances. This is very suitable for our purpose since Z3 can solve the multi-objective optimization problems directly, and we do not need to develop search strategies on top of it. To use Z3, the optimization problems were translated into the SMT-LIB format using *fzn2smt* [7] and further processed to simplify equations and reduce the number of variables. For more details we refer the reader to [33]. Note that optimization of SMT formulas is a very recent feature and still subject to a lot of research. Though it makes use of optimization techniques known from linear programming, significant progress can be expected in the future from which Zephyrus2 will directly benefit.

4 Experimental Results

In this section we describe the performance of Zephyrus2 while using different settings and solving engines.

To the best of our knowledge, due to the novelty of these approaches, there are no established benchmarks for application deployment. Moreover, the first industrial problems solved by Zephyrus in [14,18,25] were not challenging, taking only a few seconds to be solved. For this reason, to compare Zephyrus2 with Zephyrus, in this work we rely on the synthetic benchmark proposed in [16, 49]. The instances of the benchmark are derived from a parametrized variant of the WordPress deployment scenario presented in [14] and partially depicted in Figure 1. This scenario was parametrized in three dimensions to allow the analysis of scalability issues: (1) the parameter `mysql_req` encodes the number of MySQL instances a WordPress requires, (2) the parameter `wp_req` encodes the number of WordPress instances the HTTP Load Balancer requires and (3) the parameter `vm_amount` represents the amount of the four different types of Amazon EC2 virtual machines that can be used to deploy the application. The benchmark instances are obtained by varying the parameters `mysql_req` and `wp_req` within $\{6, ..., 12\}$, and `vm_amount` within $\{6, ..., 25\}$ (this corresponds to considering up to one hundred virtual machines). The goal for all the instances is to deploy an HTTP Load Balancer with the additional requirement that it is not possible to install two MySQL instances or two WordPress instances on the same machine.

We compare several different configurations of Zephyrus2 against the original approach of Zephyrus denoted as `zephyrus`. As solver backends for Zephyrus2 we use the lexicographic minimization approach with restarts using the solvers Chuffed [12] (`lex-chuffed`), Gecode [26] (`lex-gecode`), and Or-tools [27] (`lex-ortools`), as well as the SMT solver Z3 [20] (`smt`). We run all approaches using AMD Opteron 6172 processors and a timeout of 300 seconds (per problem instance), which is usually the time it takes to require and obtain a virtual machine from a cloud provider.

We remark that the times of `zephyrus` should just be used as an indicative measure of the original performance of the Zephyrus approach. Indeed, these times include also the time taken to generate the connections (bindings) between the components that, as better explained in Section 5, in Zephyrus2 is a task that by design is deferred to an external utility. However, the times of `zephyrus` are still significant for a comparison since the generation of the bindings in Zephyrus takes just few milliseconds and the

Solver	Solved		Timeout	Seconds
zephyrus	261	27%	719	67.81
lex-chuffed	980	100%	0	4.45
lex-gecode	**980**	100%	**0**	**2.25**
lex-ortools	975	99%	5	7.13
smt	960	98%	20	50.23

Table 1: Experimental results for all the approaches.

runtimes of zephyrus, especially for big problem instances, are greatly dominated by the time needed to prove that a found configuration, if any, is optimal.

We also performed experiments using MiniSearch with the aforementioned CP solvers as backends. However, due to a bug in MiniSearch,[4] it often returns a suboptimal solution. As Zephyrus2 and Zephyrus are supposed to provide optimal solutions only, we have excluded the MiniSearch approach in the following comparison.

A summary of the results is presented in Table 1. The columns *Solved* and *Timeout* denote the number of instances that were solved correctly and the number of instances where the solver was terminated due to a timeout, respectively. The last column gives the average time needed to solve the instances that could be solved within the timeout.

As can be seen, all approaches based on Zephyrus2 can solve almost all the benchmarks. While lex-chuffed and lex-gecode solve all, lex-ortools and smt lack only five and twenty, respectively, of the 980 benchmarks. As for the comparison with the original Zephyrus approach zephyrus, it is immediately visible that Zephyrus2 is faster and able to solve more instances, whatever solver is used as backend. While zephyrus is able to solve only 27% of the benchmarks in less than 5 minutes, lex-gecode which resulted as the best solver in average, is able to solve all benchmarks in at most 25 seconds (with only 10 benchmarks taking more than 10 seconds to be solved). Surprisingly, Gecode is more efficient than both Chuffed and Or-tools which, based on the results of the last MiniZinc Challenge [39] – an annual competition of constraint programming solvers – are among the best CP solvers available today. We believe that this is probably due to the nature of the deployment problem that favors the pure propagation and search approach used by Gecode. As we will see later, although smt does not manage to solve all the benchmarks and it is much slower than the other approaches of Zephyrus2, it is not dominated by lex-chuffed or lex-gecode and for a few hard instances it is faster than both of them.

Figure 2 shows for varying timeout values (x axis) the percentage of the problem instances that could be solved within that timeout (y axis) by the different approaches.

In Figure 3 we compare the individual results of the best approach lex-gecode with the other approaches. Each plot relates the benchmark results of two approaches on a logarithmic scale where every cross represents a single problem instance. Firstly we can see that lex-chuffed is almost dominated by lex-gecode as there are no examples that took more than 2 seconds in which lex-chuffed performs significantly better. The same conclusion can be drawn also for lex-ortools that is

[4] MiniSearch is a very recent framework, only available in a beta version.

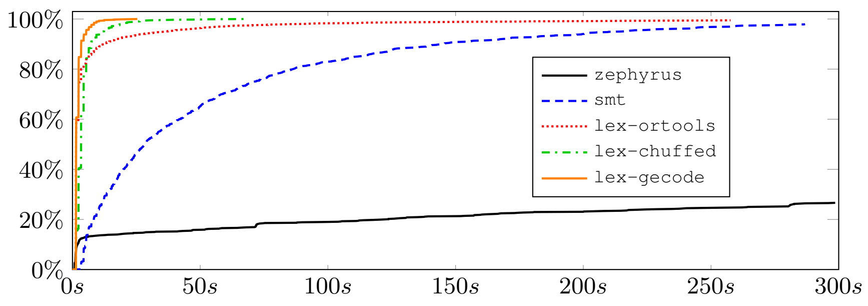


Fig. 2: Percentage of the instances that could be solved within a given timeout.

sometimes faster than `lex-gecode`, but only on comparably easy instances that are quickly solved by both solvers. In contrast to that, we believe that `smt` can be a valuable complement to `lex-gecode`, as it performs better on exactly those benchmarks that `lex-gecode` struggles to solve. For every benchmark that takes `lex-gecode` more than 10 seconds to be solved, `smt` is faster than `lex-gecode`. We conjecture that this is due to the fact that the dynamic search heuristics of the `smt` approach are more robust than the ones used by the `lex-gecode` for this problem type. A deeper comparison between these two approaches on harder instances is left as future work. Surprisingly, some of the instances that took `lex-gecode` more than 10 seconds to be solved are instead solved by Zephyrus in a shorter time. We conjecture that for these instances the search heuristics used by Zephyrus lead to a good solution faster, thus allowing to prove the optimality in shorter time.

5 Discussion

Zephyrus was the first tool to tackle the deployment optimization problem as defined in Section 2. The development of Zephyrus2 was triggered by experiences in applying Zephyrus in industrial case studies [14, 18, 25]. In this section, we discuss in detail the technical differences between Zephyrus2 and Zephyrus [16].

Modeling. Whereas Zephyrus directly uses the Aeolus [17] component model for the cloud, this component model has been extended for Zephyrus2 in order to model locations and components more naturally. A concern that arose in industrial case studies was that components may expose different interfaces at the same time. In Aeolus this is modeled by one provide-port for every interface. However, this encoding leads to an exponential blow-up in the number of components when capacity constraints are associated to the provide-port. To avoid this exponential blow-up, Zephyrus2 allows provide-ports to have one or more provided functionalities. For instance, assuming that a new version of a server is able to provide its functionalities to at most two clients via a protocol a or b, in Zephyrus2 we can specify this by adding only one provide-port offering concurrently the possibility to use both protocols to at most 2 clients. Conversely, Zephyrus requires the introduction of different components: a component offering to two clients the functionality using protocol a, a component offering to two clients the

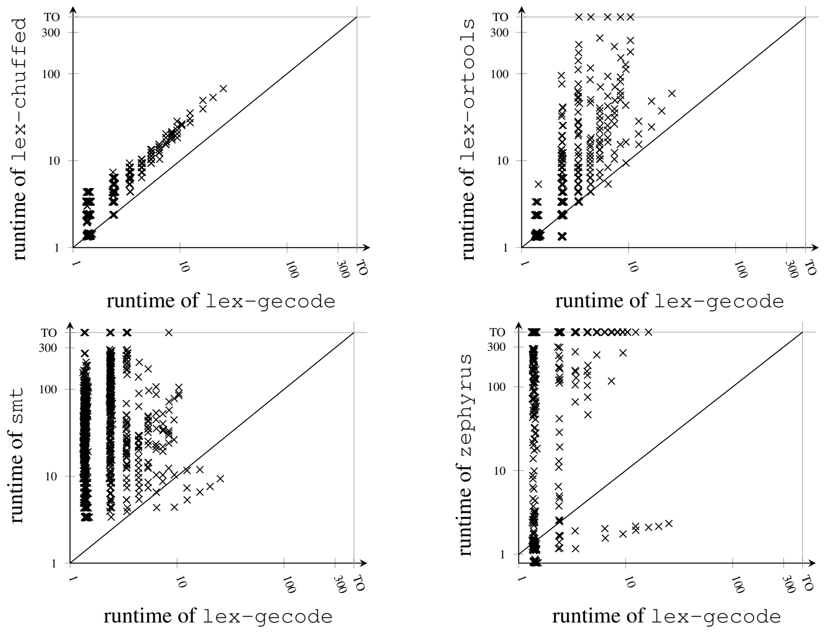


Fig. 3: Comparison of `lex-gecode` to the other approaches.

functionality with protocol b, and a component offering to one client the functionality with protocol a and to another client the functionality with protocol b.

In the deployment constraint language, support for quantifiers and sum terms (cf. Section 3) allows Zephyrus2 to express properties in a more concise way. In addition, Zephyrus2 supports user-defined metric functions to better customize the optimization, whereas users of Zephyrus are restricted to a predefined set of metrics.

Constraint solving. Zephyrus solves deployment optimization problems by translating them into a COP encoded in MiniZinc [38] and then uses standard CP solvers (such as the Gecode [26] and G12 solvers [47]) to iteratively minimize a number of cost functions through sequential invocations of an optimizing solver. Zephyrus2 offers more possibilities for the choice of solving technologies, as discussed in Section 4.

Another major difference between the tools is that Zephyrus2 relies on a completely new MiniZinc model[5], which enables Zephyrus2 to better exploit the symmetries of the deployment problem. Zephyrus initially has to consider a large number of locations to ensure that there are enough inexpensive virtual machines available. Every additional location extends the search space considerably. To cope with this problem, Zephyrus relies on ad-hoc location trimming heuristics [49] which tries to reduce the number of locations. Instead of relying on ad-hoc heuristics, it is also possible to use symmetry breaking constraints [46] to reduce the search space by removing some symmetric solutions (unless a user has machine-specific constraints on where to deploy components such as, for example, that WordPress needs to be installed on `c3_large_2` but not on

[5] The MiniZinc model, available from [33], was submitted to the MiniZinc Challenge 2016.

the similar VM `c3_large_1`). For this purpose, Zephyrus2 uses well-known symmetry breaking constraints for the bin packing problem [44]. In particular, Zephyrus2 establishes an order between locations with the same resources and enforces the deployment of the components in the cheapest location available, following the pre-determined order on these locations. This removes some of the symmetries between the locations.

Zephyrus2 supports similar constraints to break symmetries between the components, which establish an order between components and then enforce the deployment in the location following the lexicographic order. Compared to Zephyrus2, Zephyrus only uses symmetry breaking constraints for locations, but not for the components. The effectiveness of the symmetry breaking constraints in Zephyrus2 allows the tool to reach better performance than Zephyrus, even without the use of location trimming heuristics that are effective only in a limited number of deploying scenarios. [6]

Zephyrus2 simplifies the non-linear constraints used in Zephyrus into a conjunction of linear implications, by means of encoding techniques [15]. These techniques have proven effective to increase the performance of SMT solvers and allows the use of Chuffed, which does not support the non-linear constraints of the original formalization.

We would like to emphasize that the performance of the different solvers heavily depends on the encoding of the constraints, and the addition of redundant or symmetry breaking constraints. For instance, we noticed that without the so-called Ralf's redundant constraints [49], the performance of the CP solvers degrades considerably while for SMT solving they do not have any strong impact. Conversely, the simplification of the non-linear constraints allows the use of Chuffed and improves the performance of the SMT approach, but it has no impact on the performance of Gecode or Or-tools. Symmetry breaking constraints have a huge impact on the performances of both the CP and SMT approaches. More details on these are presented in Appendix A.

Based on the use of Zephyrus in [14, 18, 25], we noticed that users often have preferences over the bindings between the components. For instance, it is often better to have bindings between co-located components and avoid configurations in which, e.g., a WordPress uses the functionalities of a MySQL deployed on another location while it could have used the functionalities of a MySQL deployed in its own location. In Zephyrus, the resolution of the deployment problem is tied to the generation of the bindings performed by means of an ad-hoc polynomial algorithm. Unfortunately, Zephyrus does not take into account preferences between bindings. For this reason, as a design choice, Zephyrus2 separates the task of computing the distribution of the components in the various locations from the task of connecting the components. Contrary to Zephyrus, the generation of the connections between the components is therefore not part of the core of Zephyrus2 and is instead deferred to an external utility. In particular, Zephyrus2 comes with a default simple bindings generation utility that maximizes the number of local bindings in few seconds (for further details, see [33]).

[6] Zephyrus2 supports machine-specific constraints by removing symmetry breaking constraints.

6 Related Work

Whereas in Section 5 we discussed the differences between Zephyrus and Zephyrus2, this section considers the deployment optimization problem addressed by these tools in a broader context.

With the increasing popularity of cloud computing, the problem of automating application deployment has recently attracted a lot of attention. Many system management tools have been developed for this purpose, including popular tools such as CFEngine [8], Puppet [31], MCollective [42], and Chef [41]. Despite their differences, such tools allow to declare the components that should be installed on each machine, together with their configuration parameters. In order to use such tools, the DevOps architect needs to know how components should be distributed and configured.

Some tools aim to compute an (optimal) configuration of a distributed system without computing the deployment steps needed to reach it. CP appears to be one of the best methods today for solving different configuration problems [46]. The structure of a configured system depends on the application domain and this knowledge is exploited to speed up the search for valid configurations. CP techniques have already been applied with success to the problem of deciding the allocation of resources in data centers and clouds [9, 28, 32, 35, 36]. Zephyrus2 relies on solver techniques similar to those adopted by these tools. Indeed, the COP problems solved by Zephyrus can be seen as an extension of the well-known bin packing problem [13] where some items, corresponding to components, have to be included into bins, corresponding to locations. However, in contrast to these approaches, Zephyrus2 not only computes the optimal allocation but also identifies the additional components that are needed to form a valid configuration. Formally, even without considering the allocation of components, the problem of deciding if there is a correct configuration is already NP-hard [15]. To the best of our knowledge, no trivial encoding exists that allows the reuse of [9, 28, 32, 35, 36] to solve the deployment optimization problem tackled by Zephyrus2.

Perhaps the most similar to our approach is ConfSolve [29], which uses a formulation based on constraints to propose an optimal allocation of virtual machines to servers and of applications to virtual machines. Similar to Aeolus [17], a declarative language is used to describe the entities (e.g., machines and services), and the deployment problem is then solved by translating the declarative specification into MiniZinc. However, in contrast to Zephyrus2, ConfSolve does not handle capacity and replication constraints, so there is no obvious representation of our benchmarks for ConfSolve. Another interesting related work is Saloon [43], where a deployment problem is described by means of a feature model extended with feature cardinalities. Saloon applies CP technologies to determine a deployment. While Saloon is able to automatically detect inconsistencies between components, it does not address the optimization problem; i.e., the solutions proposed by Saloon do not minimize the number of resources and virtual machines to be used.

Other approaches rely on a range of techniques to compute optimal component allocation. For instance, SAT solvers are used to solve network configuration problems [37], a prediction-based online approach [34] is proposed to find optimal reconfiguration policies, a genetic-based algorithm [24] is used to support the migration and deployment of enterprise software with their reconfiguration policies, and integer linear programming has been used to find energy-efficient VM configurations [48]. However, none of the tools that we are aware of allows capacity and replication constraints to be expressed, which are essential non-functional constraints for any non-trivial, scalable application. Furthermore, most of them give no optimality guarantee on the solution.

7 Conclusions

In this paper we presented Zephyrus2, a tool that is advancing the state-of-the-art by computing the *cheapest* way to deploy complex cloud applications based on declarative specifications. Optimal deployments involving up to a hundred components and virtual machines can be generated within seconds. This allows Zephyrus2 to be used in a more interactive way by the DevOps architect, who does not need to wait for minutes to inspect the proposed optimal solution and restart the computation in case she has forgotten to elicit one constraint or preference.

Zephyrus2 has already been tested in an industrial environment to check the cost optimality of currently deployed solutions and to devise the optimal allocation for deploying new components. The feedback obtained so far is positive both for the tool's usability and for its running time. As witnessed in [19], the support of a more concise language to specify user constraints and the improved performance makes Zephyrus2 a better alternative to the original version of Zephyrus.

To further improve the performance of Zephyrus2 in future work, we plan to study whether the SMT encoding can be improved and especially to consider whether SMT solvers can be extended with modules that perform propagation similar to those implemented by CP solvers. Moreover, exploiting the flexibility of MiniSearch, we plan to study local search procedures, which could be extremely useful in scenarios where the user is not interested in the optimal solution but just in finding a sufficiently good solution very quickly (e.g., in a second or less).

Based on the good results obtained by the Sunny portfolio approach [1] on the last MiniZinc Challenge, we also plan to study how the different search procedures can be combined to obtain a globally better solver. In particular, we are interested in combining the strengths of the different solvers by using, e.g., the bound sharing with restarts approach of the parallel version of the Sunny solver [2].

Finally, we are also interested in enriching the formal model behind Zephyrus2 to allow constraints on the bindings between components. This will allow configurations with more complex properties to be generated, such as the non-transferability of data between borders or the load balancing of traffic between parts of the system deployed on different data centers.

Acknowledgements

We would like to thank Andreas Schutt from NICTA (National ICT of Australia) for proposing a search annotation for the deployment problem when submitting Zephyrus2 instances to the MiniZinc Challenge 2016.

References

1. R. Amadini, M. Gabbrielli, and J. Mauro. SUNNY: A lazy portfolio approach for constraint solving. *TPLP*, 14(4-5):509–524, 2014.
2. R. Amadini, M. Gabbrielli, and J. Mauro. A multicore tool for constraint solving. In *IJCAI*, pages 232–238, 2015.
3. Amazon. AWS CloudFormation. `http://aws.amazon.com/cloudformation/`.
4. C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard Version 2.6, 2015.
5. L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2013.
6. N. Bjørner, A.-D. Phan, and L. Fleckenstein. νZ - An optimizing SMT solver. In *TACAS*, pages 194–199, 2015.
7. M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012.
8. M. Burgess. A site configuration engine. *Computing Systems*, 8(2):309–337, 1995.
9. H. Cambazard, D. Mehta, B. O'Sullivan, and H. Simonis. Bin packing with linear usage costs - An application to energy management in data centres. In *CP*, pages 47–62, 2013.
10. M. Catan, R. D. 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*, pages 1–3, 2013.
11. CenturyLink. Cloud Blueprints. `https://www.centurylinkcloud.com/blueprints/`.
12. Chuffed. The CP solver. `https://github.com/geoffchu/chuffed`.
13. E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: A survey. In *Approximation Algorithms for NP-hard Problems*, pages 46–93. 1997.
14. R. D. Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Automatic deployment of services in the cloud with Aeolus Blender. In *ICSOC*, pages 397–411, 2015.
15. R. D. Cosmo, M. Lienhardt, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Automatic application deployment in the cloud: From practice to theory and back. In *CONCUR*, pages 1–16, 2015.
16. R. D. Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, and A. Agahi. Automated synthesis and deployment of cloud applications. In *ASE*, pages 211–222, 2014.
17. R. D. Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Aeolus: A component model for the cloud. *Inf. Comput.*, 239:100–121, 2014.
18. S. de Gouw, M. Lienhardt, J. Mauro, B. Nobakht, and G. Zavattaro. On the integration of automatic deployment into the ABS modeling language. In *ESOCC*, pages 49–64, 2015.
19. S. de Gouw, J. Mauro, B. Nobakht, and G. Zavattaro. Declarative elasticity in ABS. In *ESOCC*, 2016. To appear.
20. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.
21. Docker Inc. Docker. `https://www.docker.com/`.
22. J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A deployment management system. In *PLDI*, pages 263–274, 2012.
23. Flexiant. Bento Boxes. `http://www.flexiant.com/2012/12/03/application-provisioning/`.
24. S. Frey, F. Fittkau, and W. Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *ICSE*, pages 512–521, 2013.
25. M. Gabbrielli, S. Giallorenzo, C. Guidi, J. Mauro, and F. Montesi. Self-reconfiguring microservices. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 194–210, 2016.

26. GECODE. An open, free, efficient constraint solving toolkit. `http://www.gecode.org`.
27. Google. Optimization tools.
`https://developers.google.com/optimization/`.
28. F. Hermenier, S. Demassey, and X. Lorca. Bin repacking scheduling in virtualized datacenters. In *CP*, pages 27–41, 2011.
29. J. A. Hewson, P. Anderson, and A. D. Gordon. A declarative approach to automated configuration. In *LISA*, pages 51–66, 2012.
30. Juju. DevOps Distilled. `https://jujucharms.com/`.
31. L. Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1):19–25, 2006.
32. A. Malapert, J. Régin, and J. Parpaillon. The package server location problem. In *ICORES*, pages 193–204, 2013.
33. J. Mauro. Zephyrus2. `https://bitbucket.org/jacopomauro/zephyrus2/src`.
34. H. Mi, H. Wang, G. Yin, Y. Zhou, D. xi Shi, and L. Yuan. Online self-reconfiguration with performance guarantee for energy-efficient large-scale cloud computing data centers. In *SCC*, pages 514–521, 2010.
35. L. Michel, A. A. Shvartsman, E. L. Sonderegger, and P. V. Hentenryck. Load balancing and almost symmetries for RAMBO quorum hosting. In *CP*, pages 598–612, 2010.
36. L. D. Michel, P. V. Hentenryck, E. L. Sonderegger, A. A. Shvartsman, and M. Moraal. Bandwidth-limited optimal deployment of eventually-serializable data services. In *CPAIOR*, pages 193–207, 2009.
37. S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Network Syst. Manage.*, 16(3):235–258, 2008.
38. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.
39. NICTA. MiniZinc Challenge 2015.
`http://www.minizinc.org/challenge2015/results2015.html`.
40. OASIS. Topology and orchestration specification for cloud applications (TOSCA) version 1.0. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html`.
41. Opscode. Chef. `https://www.chef.io/chef/`.
42. Puppet Labs. Marionette collective.
`http://docs.puppetlabs.com/mcollective/`.
43. C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, and G. Botterweck. Consistency checking for the evolution of cardinality-based feature models. In *SPLC*, pages 122–131, 2014.
44. J. Régin and M. Rezgui. Discussion about constraint programming bin packing models. In *AI for Data Center Management and Cloud Computing*, pages 21–23, 2011.
45. A. Rendl, T. Guns, P. J. Stuckey, and G. Tack. MiniSearch: A solver-independent meta-search language for MiniZinc. In *CP*, pages 376–392, 2015.
46. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.
47. P. J. Stuckey, M. G. de la Banda, M. Maher, K. Marriott, J. Slaney, Z. Somogyi, M. Wallace, and T. Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *ICLP*, pages 9–13, 2005.
48. P. N. Tran, L. Casucci, and A. Timm-Giel. Optimal mapping of virtual networks considering reactive reconfiguration. In *CLOUDNET*, pages 35–40, 2012.
49. J. Zwolakowski. *A Formal Approach to Distributed Application Synthesis and Deployment Automation*. PhD thesis, Université Paris Diderot Paris 7, 2015.

A Performances for Varying MiniZinc Formalizations

In this section, for the interested reader, we present the performances of the aforementioned approaches `lex-gecode`, `lex-ortools`, `lex-chuffed`, and `smt` when using different variants of the COP model. In particular we show the results for the benchmarks described in Section 4

1. without the so called Ralf's redundant constraints [49],
2. without symmetry breaking constraints, and
3. if we use non-linear constraints in the encoding.

The model is changed only for Zephyrus2, i.e. for `lex-chuffed`, `lex-gecode`, `lex-ortools` and `smt`, but we also report `zephyrus` results as a baseline.

Redundant constraints [46] are constraints that are implied by other constraints. Strictly speaking they are not needed but sometimes they help a solver to prune the search space faster. For example, Ralf's redundant constraints state that when a component is present that requires a port p with capacity x then there should be at least x other components providing the port p. Figure 4a presents results when Ralf's redundant constraints are not added to the encoding. In comparison to Figure 2, Figure 4a shows that the presence of these constraints makes a huge difference for `lex-ortools` and `lex-gecode` while it has no significant effect on `lex-chuffed` and `smt`. We believe that this is due to the fact that the latter are able to learn these constraints by themselves during the search while the former lack learning abilities. Thus, `lex-gecode` and `lex-ortools` can only benefit from this additional pruning if they are provided with these constraints explicitly.

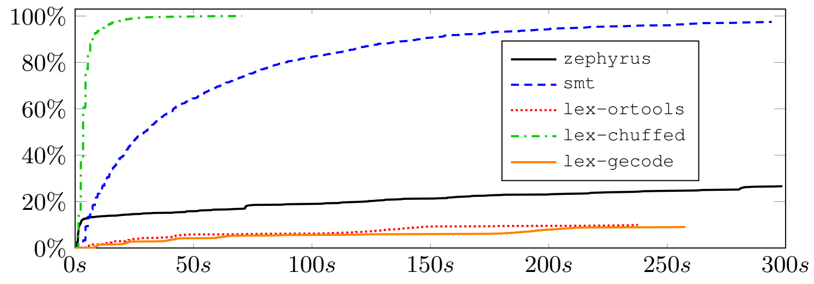
Figure 4b shows that all the approaches worsen their performances, becoming even worse than the original Zephyrus approach[7], when we do not add symmetry breaking constraints. Thus it seems that symmetry breaking constraints are highly beneficial for all tested solvers. Albeit there are heuristics to mitigate symmetries automatically, symmetry breaking constraints tailored to the specific problem prove to be much more effective.

In the previous experiments the encodings were linear. However, there are also natural and concise non-linear formulations. For example, given two components x and y, the fact that there could not be two bindings connecting the port p between an instance of x and an instance of y has the necessary non-linear condition that $\texttt{bind}(p,x,y) \leq \texttt{comp}(x) \times \texttt{comp}(y)$. As proven in [15], this constraint can also be stated as

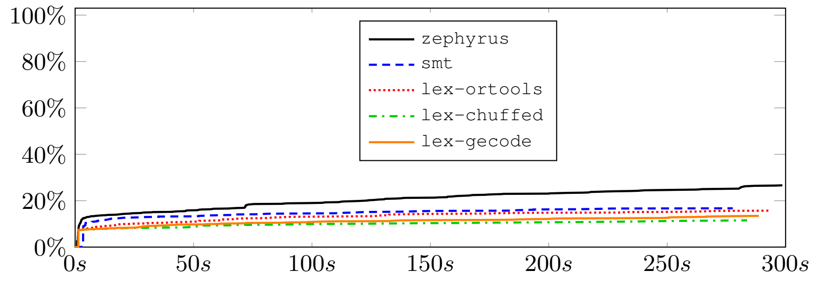
$$\texttt{comp}(x) \geq r \rightarrow \texttt{bind}(p,x,y) \leq r \times \texttt{comp}(y)$$
$$\bigwedge_{i<r} (i = \texttt{comp}(x) \rightarrow (\texttt{bind}(p,x,y) \leq i \times \texttt{comp}(y)))$$

where r is the number associated to the required port p of the component y. Figure 4c shows that the occurrence of non-linear constraints does not tamper with the performances of the standard CP solvers Or-tools and Gecode but it hinders the SMT solver.

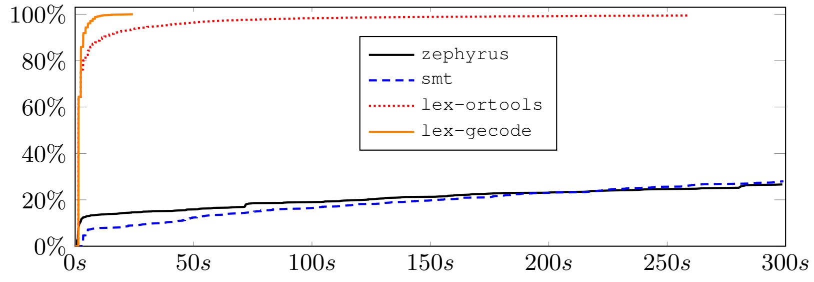
[7] Even when the user have machine-specific constraints, it is still possible via a simple encoding to exploit symmetry breaking for locations without any machine-specific constraints.



(a) No Ralf's redundant constraints.



(b) No symmetry breaking constraints.



(c) Non-linear encoding.

Fig. 4: Percentage of the instances that could be solved within a given timeout for different encodings.

We conjecture that this is due to the fact that while the CP solver can deal with the non-linear constraints natively, the SMT solver instead has to linearize them as it does not support non-linear optimization yet. It seems that our direct encoding that exploits some domain knowledge is better than the more general translation performed by the SMT solver. Please note that in Figure 4c we do not present `lex-chuffed` because it does not support non-linear optimization.

B Excerpt of the MiniZinc encoding

For Zephyrus2, the problem specification is divided in two parts. The first part, shared by all the backends for all problem instances, defines the constraints that enforce the consistence of the configuration, the symmetry breaking constraints and the redundant constraints. The second part instead adds the constraints to specify what to optimize and the user constraints. This part of the specification depends on the instance to be solved and also from the chosen backend. Additionally, Minizinc data files are used to specify the locations and the components.

In the following we show the first part of the specification. An example of the second part of the specification for an instance of the WordPress scenario described in Section 4 and the `lex-gecode` is shown at the end of this section.

```
include "lex_greatereq.mzn";
include "lex_less.mzn";

int: MAX_INT = 1024;

%%%%%%%%%%%%%%%%
% Input parameters
%%%%%%%%%%%%%%%%

% components
set of int: comps;
% ports
set of int: ports;
% multiple provide port
set of int: multi_provide_ports;

% locations
set of int: locations;
% resources
set of int: resources;

% map of components with their requirements number
array[ comps, ports] of int: requirement_port_nums;
% map of components with their provided multi-ports
% -1 means infinite multi-port provider
array[ comps, multi_provide_ports] of int: provide_port_nums;
% map of components with their conflicts
array[ comps, ports] of bool: conflicts;
% map of multi-ports with their ports
array[ multi_provide_ports, ports] of bool: multi_provides;

% map of location with their costs
array[ locations ] of int: costs;
% map of locations with the resouces they provide
array[ locations, resources ] of int: resource_provisions;
% map of components with the resources they consume
array[ comps, resources ] of int: resource_consumptions;

%%%%%%%%%%%%%%%%
% variables
%%%%%%%%%%%%%%%%
```

```

% bindings number
array[ multi_provide_ports, ports, comps, comps] of var 0..
    MAX_INT: bindings;
% components number
array[ comps ] of var 0..MAX_INT: comps_num;
% location to number of component map
array[ locations, comps] of var 0..MAX_INT: comp_locations;

% total number of components
var 0..MAX_INT: sum_comp;

%%%%%%%%%%%%%%%%
% constraints (no location)
%%%%%%%%%%%%%%%%

% bind the total number of components
constraint sum_comp = sum( i in comps)(comps_num[i]);

% bindings 0 if the multiprovide does not provide port
constraint forall(mport in multi_provide_ports, port in ports) (
  if multi_provides[mport,port]
  then true
  else forall(i in comps, j in comps) ( bindings[mport,port,i,j]
       = 0)
  endif
);

% provides must be greater or equal to bindings & infinite
    provide port constraints
constraint forall(mport in multi_provide_ports, pcomp in comps)
    (
  if provide_port_nums[pcomp,mport]=0
  then forall(port in ports, rcomp in comps) (
    bindings[mport,port,pcomp,rcomp] = 0
  )
  else
    if (provide_port_nums[pcomp,mport] = -1)
    then forall(port in ports, rcomp in comps) (
     (comps_num[pcomp] = 0) -> (bindings[mport,port,pcomp,rcomp
         ]=0))
    else sum( port in ports, rcomp in comps)(
      bindings[mport,port,pcomp,rcomp] ) <= comps_num[pcomp] *
provide_port_nums[pcomp,mport]
    endif
  endif
);

% requires must be equal to bindings
constraint forall(port in ports, rcomp in comps) (
  sum( mport in multi_provide_ports, pcomp in comps)(
    bindings[mport,port,pcomp,rcomp] ) = comps_num[rcomp] *
requirement_port_nums[rcomp,port]
);


% conflict constraints if component provide same port
constraint forall(port in ports, pcomp in comps) (
  if (conflicts[pcomp,port] /\ exists(mport in
      multi_provide_ports) (
    (multi_provides[mport,port]) /\ provide_port_nums[pcomp,
        mport] != 0))
  then comps_num[pcomp] <= 1
  else true
  endif
);

% conflict constraints
constraint forall(port in ports, pcomp in comps, rcomp in comps)
     (
  if (conflicts[rcomp,port] /\ exists(mport in
      multi_provide_ports) (
    (multi_provides[mport,port]) /\ provide_port_nums[pcomp,
        mport] != 0))
  then comps_num[pcomp] > 0 -> comps_num[rcomp] = 0
  else true
  endif
);

% unicity constraint
% note that we require that a component does not
% require more than one port provided by a
% multiple provide port
constraint forall(mport in multi_provide_ports, pcomp in comps,
    rcomp in comps)
(
  let
      { int: max_req = sum(port in ports) (
          if multi_provides[mport,port]
          then requirement_port_nums[rcomp,port]
          else 0
          endif
        )
      } in
      if pcomp = rcomp
      then
        ( comps_num[pcomp] >= max_req  ->
          sum(port in ports)(bindings[mport,port,pcomp,rcomp])
            <= max_req * (comps_num[rcomp] - 1))
        /\
        ( comps_num[pcomp] < max_req  ->
          forall (i in 1..max_req)( comps_num[pcomp] = i ->
            sum(port in ports)(bindings[mport,port,pcomp,rcomp]) <= i * (i-1) ))
      else
        ( comps_num[pcomp] >= max_req  ->
          sum(port in ports)(bindings[mport,port,pcomp,rcomp])
            <= max_req * comps_num[rcomp] )
        /\
        ( comps_num[pcomp] < max_req  ->
          forall (i in 0..max_req)( comps_num[pcomp] = i ->
            sum(port in ports)(bindings[mport,port,pcomp,rcomp]) <= i * comps_num[rcomp] ) )
      endif
);

%%%%%%%%%%%%%%%%
% constraints for deciding locations
%%%%%%%%%%%%%%%%

% map location used or not used
array[ locations ] of var 0..1: used_locations;
constraint forall( l in locations)(
  sum(c in comps)(comp_locations[l,c]) = 0 <-> used_locations[l] = 0
);

constraint forall( c in comps) (
  sum( l in locations) ( comp_locations[l,c]) = comps_num[c]
);

constraint forall( res in resources, loc in locations) (
  sum( comp in comps)( comp_locations[loc,comp] *
resource_consumptions[comp,res] )
    <= resource_provisions[loc,res]
);

% the number of locations can not be greater than the number of components
% i.e, one component per location in the worst case
constraint sum(i in locations) (used_locations[i]) <= sum_comp;

%%%%%%%%%%%%%%%%
% symmetry constraints (wrap in a unique predicate following minizinc specs)
%%%%%%%%%%%%%%%%

predicate symmetry_breaking_constraints( bool: b) =
  %if location are equal then first location has more comps than the second
  %in lexicographically order (based on the cost)
  forall( l1 in locations, l2 in locations)(
    if l1 < l2 /\ forall ( r in resources)(resource_provisions[l
        1,r] = resource_provisions[l2,r] )
    then
      if costs[l1] > costs[l2]
      then lex_greatereq(
        [comp_locations[l2,c] | c in comps],
        [comp_locations[l1,c] | c in comps])
        /\
        (used_locations[l1] = 1 -> used_locations[l2] = 1)
      else
        lex_greatereq(
          [comp_locations[l1,c] | c in comps],
          [comp_locations[l2,c] | c in comps])
          /\
          (used_locations[l2] = 1 -> used_locations[l1] = 1)
      endif
    else
      true
    endif
  );

%%%%%%%%%%%%%%%%
% redundant constraints
%%%%%%%%%%%%%%%%

predicate redundant_constraints(bool: b) =
  % Ralf constraints
  forall (rcomp in comps, port in ports)(
    if requirement_port_nums[rcomp,port]!=0
    then
      comps_num[rcomp] > 0 -> sum( mport in multi_provide_ports,
           pcomp in comps)(
      if provide_port_nums[pcomp,mport] != 0 /\ multi_provides[
          mport,port]
      then comps_num[pcomp]
      else 0
      endif ) >= requirement_port_nums[rcomp,port]
    else
      true
    endif
  );

constraint redundant_constraints(true);

%%%%%%%%%%%%%%%%
% solved items
%%%%%%%%%%%%%%%%

```

```
int: obj_min = sum(l in locations)(if costs[l] < 0 then costs[l]
     else 0 endif);
int: obj_max = sum(l in locations)(if costs[l] > 0 then costs[l]
     else 0 endif);
var obj_min..obj_max: total_cost;
constraint total_cost = sum(l in locations)(used_locations[l] *
    costs[l]);

array [locations] of locations: costs_asc  = arg_sort(costs);
array [locations] of locations: costs_desc = reverse(costs_asc);
```

In the following we show the second part of the specification defining the user constraints, and the search strategy used by `lex-gecode` to minimize the cost of deploying one WordPress scenario as defined in Section 4.

```
constraint (sum ( l in locations) ( comp_locations[l,3])>0 \/
  sum ( l in locations) (
    comp_locations[l,2])>0)
    /\ (forall(x in locations )(comp_locations[x,1]<2 ))
    /\ (forall(x in locations )(comp_locations[x,4]<2 ));

constraint symmetry_breaking_constraints(true);

array[1..2] of var int: obj_array;
constraint obj_array[1] = total_cost;
constraint obj_array[2] = (sum(y in comps )
  (sum(l in locations)(comp_locations[l,y]) ));
solve :: seq_search([
              int_search([used_locations[costs_desc[i]] | i in
                  locations],
                input_order, indomain_min, complete),
              int_search([comp_locations[l, i] | l in locations,
                   i in comps],
                first_fail, indomain_max, complete),
              int_search(comps_num, first_fail, indomain_max,
                  complete),
              int_search([bindings[m,p,i,j] | m in
                  multi_provide_ports,
                p in ports, i,j in comps],
                first_fail, indomain_max, complete)
              ])  minimize obj_array[1];
```