A formal approach to distributed application synthesis and deployment automation

Jakub Zwolakowski

▶ **To cite this version:**

Jakub Zwolakowski. A formal approach to distributed application synthesis and deployment automation. Software Engineering [cs.SE]. Univeristé Paris Diderot Paris 7, 2015. English. NNT : . tel-01172022

HAL Id: tel-01172022
https://theses.hal.science/tel-01172022

Submitted on 6 Jul 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Paris Diderot – Paris 7
Sorbonne Paris Cité

École Doctorale Sciences Mathématiques de Paris Centre

Laboratoire Preuves, Programmes et Systèmes (PPS)

DOCTORAT
INFORMATIQUE

Jakub Zwolakowski

A formal approach to distributed application synthesis and deployment automation

Une approche formelle à la synthèse des applications distribuées et leur déploiement automatique

---

Thèse dirigée par
Roberto Di Cosmo et **Stefano Zacchiroli**
Soutenue le 23 Avril 2015

Jury

M.	Jean-Bernard Stefani	:	*Président*
M.	Roberto Di Cosmo	:	*Directeur*
M.	Stefano Zacchiroli	:	*Co-directeur*
Mme	Laurence Duchien	:	*Rapporteuse*
M.	Maurizio Gabbrielli	:	*Rapporteur*
M.	Paul Anderson	:	*Examinateur*
M.	Manuel Carro	:	*Examinateur*
M.	François Armand	:	*Examinateur*

CONTENTS

ACKNOWLEDGEMENTS

Research

First of all, I would like to enormously thank my supervisor, **Roberto Di Cosmo**. He pushed and pulled me through this thesis in his calm, gentle and respectful way, usually managing to give me a nudge exactly when I needed one to keep advancing. And he successfully guided me to this happy ending :) I never really understood how he managed to take care of me while simultaneously being responsible for around 5 million more important things (directing IRILL being only one of them), but somehow he was able to find time to see me regularly and he was always there for me at short notice when I needed a rescue in some unexpected administrative urgency. Grazie mille Roberto!

Second of all, I would like to thank very much my co-supervisor, **Stefano Zacchiroli**. I really appreciate the time and effort which he put into reading my whole bloated thesis manuscript from the beginning to the end so carefully (I cannot even imagine how he must have suffered during this task...) and hunting down all the mistakes and issues big and small. Also I would like to thank him in particular for being patient enough to only look at me with mild irritation, instead of ripping my head off straight away, when on multiple occasions I showed complete ignorance of important topics (especially those related to free software) or just said something plainly stupid ;) Grazie mille Stefano!

Third of all, I would like to immensely thank **Ralf Treinen**. Considering his importance in the context of my PhD studies he should probably be appointed the position of my honorary second co-supervisor :) From the strictly scientific point of view, his contributions to the research on which my thesis is based are undeniably substantial. From a more human point of view, his constant interest in my advancements and the amount of direct help he provided me in various situations were very valuable. All that not even mentioning the fact that without him I would most probably never have become a PhD student... Danke schön Ralf!

Fourth of all, I would like to thank a lot **Michael Lienhardt**. It were his brilliant ideas which made many parts of the approach presented in this thesis actually possible (long live binding unicity constraints!) and it was his talent for clearly formalizing stuff which made implementing this approach so straightforward. Also I would like to thank him for all our fruitful face-to-face discussions after the team meetings, which led to concretizing so many of these concepts. Merci beaucoup Michael! (Je dois aussi avouer que pendant certain temps peut-être je te détestais un petit peu pour intervenir si brutalement dans mon code... Mais je t'ai pardonné. Et c'est ça qui compte à la fin ;))

Also I would like to thank all the people from the other Aeolus project teams:

- **Antoine Eiche** from Mandriva, author of the Armonic tool, who almost single handedly did the extremely hard work of implementing all the elegant abstractions of our model in practice, and thus proved that the model itself (together with the whole approach) is worthwhile and can be successfully applied in the real world.

- The whole **Bologna** team: **Gianluigi Zavattaro**, **Jacopo Mauro** and **Tudor Lascu**, very competent and hard working people, who contributed so much to many things on which the work presented in this thesis is based.
- As for the Aeolus team from **Nice Sophia Antipolis** I will restrain myself here to saying: thank you **Jean-Charles**, **Arnaud** and **Mohamed**. You are nice people :) And I am sure that one day you will find an actually motivating research project (i.e. one which involves more *bin packing* than this one, I assume).

Also I would like to thank **Alexis Agahi** for his faith in the Aeolus approach and the Zephyrus tool and his great motivation in introducing it and adopting in practical situations in the Kyriba Corporation.

Jury

I would like to express my sincere gratitude to those who have kindly agreed to be in my defence jury: especially big thanks are due to **Laurence Duchien** and **Maurizio Gabbrielli** for taking on their shoulders the heavy duty of reading my manuscript in detail and writing the official reviews; thanks to **Jean-Bernard Stefani** for accepting the responsibility of becoming the jury's president; thanks to **Paul Anderson** and **Manuel Carro** for travelling from afar in order to assist in this event; and finally thanks to **François Armand** for sacrificing his valuable time to be here.

These who had an impact on my thesis and / or who contributed to my life during the thesis

I would like to infinitely thank **Maud**, who supported me and had to put up with me through around 97% of my PhD studies :) Without her I would certainly not have come to Paris and not have studied at Paris 7 in the first place. Also I would not have survived the French bureaucracy, not have learned to speak and write French so well and so quickly, and maybe even not have decided to accept the offer of beginning this thesis. Merci beaucoup Maud! Pour ça et pour toutes les autres choses.

PPS and around

My life in the **PPS laboratory** during these three years and half has been a bliss. I would like to thank everybody who has contributed to the great scientific and personal atmosphere which I have experienced here.

D'abord **Odile** mérite ici un remerciment spécial. Je ne sais pas si elle se rend bien compte à quel point c'est important d'avoir une personne à la fois compétente et sympathique (et n'essayant pas d'abuser son pouvoir!) qui s'occupe de choses administratives, mais elle est exactement la personne qu'il faut :) Elle a rendu beaucoup des aspects de ma vie dans le labo tranquille et facile pendant toute ma thèse et j'en suis énormément reconnaissant. Merci beaucoup Odile!

Without question the most special thanks go to **Shahin** and **Ioana**, with whom I have shared the best office ever. Though Shahin is (surprisingly) not the best person in the world, if he just put some effort and got his act together he might very well get there ;) Either way, he is really great and I am very happy that he was there during these years to let us bask in his awesomeness and incidentally organize the social life all around :) And big thanks to Ioana, who is a smart, cool, sweet person with a good heart and an admirable self-discipline. She is also a little bit our adopted child (with Shahin). However, although we have been sincerely trying very hard, we did not succeed to mold her in our image... maybe for the best :)

Special thanks to **Lourdes**, **Guillaume**, **Jehanne** (even though she is from LIAFA) and **Juliusz**, who were always around. I had a lot of great moments with them, both in the lab and outside :)

And very special thanks to **Jovana**. Maybe she has not been there for a long time, but somehow she quickly became very important ;) And she should keep in mind that now, as she has taken my place literally, she has a certain moral responsibility to take it also figuratively!

Thanks to all these who are working here now, both fellow PhD students, post-docs, *les permanents* and interns: **Étienne**, **Matthieu**, **Flavien**, **PiM**, **Giulio**, **Hadrien** (who should *really* read *Winnie the Pooh*!), **Pierre B.**, **Thibaut**, **Béatrice**, **Pablo** (and **Elisa**!), **Marie**, **Amina**, **Ludovic**, **Pierre V.**, **Cyrille**, **Christine** (*car que ferions-nous sans elle?*), **Yann**, **Jean**, as well as **Bruno**, **François** (both from LIAFA), and **Florent** (technically from the 7th floor, but I have only seen him in PPS ;)).

Thanks to all those who were here before and left in the meantime: **Guillaume** (Munch-Maccagnoni), **Fabien**, **Gabriel** (the first PhD defence in PPS that I actually understood!), **Alexis**, **Stéphane** (but these thanks are only for the period *after* he finally stopped wearing *that* pullover ;)), **Sergueï**, **Jonas**, **Antoine** and **Mehdi**.

Also thanks to the other PPS dwellers who I never actually had a longer conversation with (i.e. other than "Bonjour! – Bonjour!"), but who still looked great hanging out in the background and / or discussing their research with each other :)

Additionally, special thanks are due to the members of the honourable **FFSPPS** initiative. Especially: **Bingo**, **Hermes**, **Bianca**, **Kentucky**, **Nelson**, **Jinx**, and others. Obviously for the reasons of personal security they must remain anonymous, hence their real names will not be revealed here. *Baldy halal no!*

Not PPS

Huge thanks to **Samy** and **Rafa**, my two favourite logicians. And my two favourite Americans. And both among my absolutely favourite people :) I am really really really happy that we became friends.

And thanks to all the other friends: **Jean-Philippe**, **Benoît**, **Seb**, **Nico**, **Luis** and **Shadi**, and **Nadja**. Thanks to the remaining members of the legendary *Les Trois Colloques* who I have not mentioned yet: **Romain** and **Tristan**. And of course thanks to the *new* colloque (who is also my flat-mate now): **Nikita**. And thanks to **Lillian** and **Garrett**. I have greatly enjoyed spending time with each of you :)

And, obviously, thanks to **Ronimo Games** for the awesome Awesomenauts ;)

Last but not least

I would also like to thank my best friends back home: **Adam** and **Kasia**. The week of sailing we did each summer, and in fact any amount of time I spent with them, always helped me to clear my mind, restore my energy and regain my balance. Dzieki Kasiu i Adasiu!

Finally, I would like to thank very much my mum and my stepfather, who both came to Paris for this defence. My mum is constantly in touch with me and she has been supporting me all the time during my thesis. Dziekuje Ci mamo za ciagle wsparcie! I dziekuje Ci Jacku, że przyjechales!

Those I forgot about

To all my friends, colleagues, and other people who I should have mentioned here but forgot: thank you!

Abstract

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

In this work, we describe the different variants of the Aeolus formal model and the Zephyrus tool developed during the Aeolus research project, which attempts to tackle the scientific challenges related to the administration of modern distributed systems and solve them in practice. The various parts of the Aeolus model allow us to express a high-level specification of the desired system, the set of available components together with their requirements, and the maximal amount of virtual resources to be committed. The associated Zephyrus tool, based on this formal model, is capable of automating the assembly of complex distributed applications: it synthesizes the full architecture of the system, placing components in an optimal manner using the minimal number of available machines. Other tools developed in the Aeolus project can then be used to automatically deploy the complete system in a cloud environment.

Résumé

Des applications reparties complexes sont assemblées en connectant des composants logiciels distribués sur plusieurs machines. Concevoir et déployer tels systèmes est un problème difficile, qui nécessite un niveau important d'expertise: l'architecte système doit s'assurer que toutes les dépendances des composants sont satisfaits, éviter les composants incompatibles et ajuster la quantité de répliques du chaque composant pour tenir compte de la qualité de service et la tolérance aux pannes. Dans un environnement cloud il faut aussi minimiser les ressources virtuelles utilisées, afin de réduire le coût. Une fois l'architecture complète est conçue, il est nécessaire d'orchestrer correctement le processus de déploiement, en assurant que tous les composants sont installés, démarrés et connectés dans le bon ordre.

Dans ce travail, nous décrivons le modèle formel Aeolus et l'outil Zephyrus, développés au cours du projet de recherche Aeolus, qui tente d'aborder les défis scientifiques liés à l'administration des systèmes distribués modernes et de les résoudre en pratique. Le modèle Aeolus nous permet d'exprimer une spécification de haut niveau du système souhaité, l'ensemble des composants disponibles avec leurs exigences et la quantité maximale de ressources virtuelles à engager. L'outil Zephyrus, basé sur ce modèle formel, est capable d'automatiser l'assemblage des applications distribuées complexes: il synthétise l'architecture complète du système en plaçant les composants de manière optimale sur les machines disponibles. Autres outils développés dans le projet Aeolus peuvent ensuite être utilisés pour déployer automatiquement le système complet dans un environnement cloud.

CHAPTER 1

RÉSUMÉ

Le paysage informatique a subi un changement radical depuis l'arrivée de la première génération d'offres technologiques connues collectivement comme Cloud Computing. Même si ce terme est loin d'être précis, la promesse générale de la technologie sous-jacente est d'offrir des ressources informatiques (par exemple la puissance de calcul, l'espace de stockage) dans une façon flexible et moins coûteuse: les ressources peuvent être louées à la place d'être acquises dès le départ, et peuvent rapidement être mises à l'échelle (agrandies ou diminuées) en fonction des besoins de l'utilisateur final.

Mais des défis importants doivent encore être confrontés. Après avoir navigué à travers l'*hype* initiale, typique pour les premières années de chaque nouvelle technologie, les utilisateurs ont découvert que la possibilité de louer des dizaines de milliers de machines virtuelles pendant quelques semaines par année n'est qu'une première étape. Le déploiement, la configuration, la maintenance et l'évolution des applications et systèmes qui fonctionnent sur ces machines virtuelles nécessitent toujours des mois de travail des administrateurs et ingénieurs système hautement qualifiés et coûteux.

Cette thèse contribue à automatiser une partie de ce travail, en présentant une approche innovatrice à la synthèse des architectures basées sur des composants distribués, particulièrement pertinentes pour le développement et déploiement des applications dans le Cloud.

Le travail présenté dans cette thèse a été réalisé dans le cadre d'Aeolus, un projet de recherche et développement financé par l'ANR qui a impliqué le laboratoire PPS de l'Université Paris Diderot, le laboratoire I3S de l'Université de Nice-Sophia Antipolis, et le projet Inria Focus. Le projet Aeolus a été lancé en 2010 et a pris fin en 2014, il a été organisé autour de trois axes de recherche principaux:

Tâche 1 Concevoir un **modèle abstrait** de Cloud, qui englobe les paquets logiciels installés sur chaque machine (virtuelle), ainsi que les dépendances dynamiques entre les logiciels, machines et services.

Tâche 2 Concevoir un puissant **langage des requêtes de reconfiguration de haut niveau** permettant d'exprimer des sophistiquées demandes de reconfiguration de logiciels fonctionnant sur des groupements de machines (virtuels).

Tâche 3 Développer des algorithmes de solveurs spécialisés, permettant de **répondre efficacement aux requêtes de reconfiguration** en les traduisant en **plans de reconfiguration de bas niveau**, similaires à des actions mises en œuvre par des outils populaires de déploiement de logiciel.

Dans le cadre de ce programme de recherche, notre travail est spécifiquement axé sur la modélisation des applications distribuées, la synthèse des architectures des applications à partir d'un langage de spécification de haut niveau à travers un encodage dans des contraintes numériques, et la mise en œuvre de ces idées dans un outil sophistiqué et flexible, Zephyrus, qui est maintenant utilisé dans le cadre d'une chaîne complète d'outils mis au point par la société Mandriva[1].

[1] `http://www.mandriva.com/`

1.1 Le travail fait

Mon implication dans le travail lié avec le projet Aeolus a commencé lors de la première année de mon doctorat, en aidant dans la recherche sur l'état de l'art dans les technologies de Cloud existants. Nous étions intéressés à la fois par les solutions industrielles employées en pratique et aussi par toutes les études scientifiques potentiellement applicables à la gestion de configuration de Could et à l'automatisation de déploiement. Notre objectif était de comprendre quel est exactement le niveau actuel des connaissances théoriques et du savoir-faire pratique dans les différents domaines touchant aux objectifs du projet Aeolus, discerner les approches et les idées les plus pertinentes dans notre contexte particulier, qui pourraient nous inspirer ou sur lesquels nous pourrions nous appuyer, et de déterminer où les meilleures possibilités d'amélioration se présentent.

La partie de cette recherche à laquelle j'ai contribué le plus se portait sur quelques populaires et amplement utilisés outils de gestion de configuration distribuée de l'état de l'art. J'ai effectué une étude approfondie concernant un ensemble limité des outils de ce genre, en essayant d'atteindre plusieurs objectifs:

- analyser les solutions existantes qui marchaient bien et les cas d'utilisation typiques qu'elles arrivent à aborder, afin de mieux comprendre la problématique de la gestion de configuration des logicielles distribuées, et aussi pour reconnaître des difficultés qui apparaissent habituellement dans ce contexte et les méthodes typiques utilisées pour les résoudre;
- extraire et comparer les *modèles*, implicites ou explicites, utilisés dans chacun de ces outils pour représenter le système distribué, sa configuration et les changements de cette configuration;
- évaluer ces outils dans le contexte de l'intégration potentielle dans le projet Aeolus comme le backend du déploiement de bas niveau.

Cette étude, qui a mené à la rédaction du rapport technique "Distributed System Administration Technologies, a State of the Art" [125], est décrite plus en détail dans le chapitre 3.

Toutes ces premières enquêtes nous ont aidé considérablement à établir le *modèle Aeolus*: un modèle à composants, simple or expressive, visant à décrire formellement les systèmes distribués modernes et leurs reconfigurations du point de vue administratif, sur le niveau des services qui fonctionnent dans le système et des relations entre eux. Ce modèle a été inspiré par multiples idées prises de diverses sources (voir le chapitre 9 pour plus de détails) et développé en tenant compte nos connaissances sur les exigences typiques liées avec l'administration du système et l'état de l'art dans les technologies de gestion de la configuration.

Le modèle Aeolus est basé sur un compromis fondamental entre l'expressivité et la simplicité. Il a été conçu pour inclure le strict minimum nécessaire pour modéliser toutes les fonctionnalités que nous avions jugé essentiels: les dépendances et les conflits entre les services, le cycle de vie des services, certaines contraintes non-fonctionnelles comme la redondance et la capacité, etc. Il est ainsi assez expressif pour capturer la plupart des propriétés importantes dans les vrais des systèmes distribués, tandis que dans le même temps il reste aussi simple et abstrait que possible, afin d'être adapté pour une étude formelle de la complexité des problèmes liés à l'automatisation de déploiement. Avant tout, il abstrait les détails concernant la localisation des services sur les machines (virtuelles).

Le modèle formel Aeolus et ses propriétés ont été décrites dans trois articles: *Towards a Formal Component Model for the Cloud* [36], *Component Reconfiguration in the Presence of Conflicts* [33] et *Aeolus: a Component Model for the Cloud* [34]. Quoique je n'aie pas contribué directement à ces papiers, ni au développement du modèle lui-même, dans le chapitre chapitre 4 je le présente et je discute en détail tous ses éléments et propriétés, comme il est fondamental dans l'approche décrite dans le reste de ce document.

L'étude de la complexité des reconfigurations de logiciels utilisant le modèle Aeolus a rendu des résultats assez limitants. Le formellement défini *problème de reconfiguration* (voir la section 4.5), qui consiste essentiellement à chercher un moyen de reconfigurer un système distribué donné dans l'intention d'atteindre une certaine nouvelle configuration souhaitée, s'est avéré être indécidable dans le modèle Aeolus complète (et plutôt difficile à résoudre dans toutes sauf les plus simplifiées versions du modèle). Pour procéder, nous avons donc décidé d'appliquer le principe de diviser et régner en séparant ce problème

en deux parties presque orthogonales. Ensuite, nous (ici "nous" fait référence à l'équipe du projet Aeolus situé à Paris) nous sommes concentrés sur le sous problème de la *synthèse de l'architecture* (voir la section 5.2), qui consiste à chercher une nouvelle configuration pour un système distribué donné, qui est correcte, satisfaite une certaine requête et est optimale en fonction de critères fournis. Simultanément une autre équipe de projet Aeolus (l'équipe INRIA Focus) a tenté d'attaquer le sous problème complémentaire de la planification du déploiement optimal.

Dans le chapitre 5 je présente en détail notre approche au problème de la synthèse de l'architecture basée sur les contraintes, dont j'ai participé à la conception. Cette approche se compose de trois étapes de base:

1. d'abord nous encodons une instance donnée du problème de synthèse de l'architecture sous une forme d'un ensemble des contraintes de domaine fini,
2. puis nous résolvons le problème à contraintes généré de telle manière en utilisant un solveur externe,
3. et enfin nous traduisons la solution du problème des contraintes en forme de la solution du problème initial.

L'obstacle le plus difficile que nous avons dû confronter lors de l'élaboration de cette approche était comment encoder correctement les exigences de redondance et de capacité entre les services sous une forme de contraintes, puis comment établir de quelle manière ces services devraient être liés ensemble pour assurer la correction de la solution produite. Ceci a été réalisé en utilisant une combinaison de deux idées ingénieuses: les contraintes que nous appelons *unicité des liaisons* (voir la section 5.3.3) et une méthode que nous appelons le *matching algorithm* (voir la section 5.3.4) utilisé pour générer les liaisons correctes entre les instances de services.

Tous les éléments de cette approche basée sur les contraintes ont été formalisés et prouvés corrects. Une variante spéciale du modèle Aeolus a été introduite à cette fin, ensemble avec une syntaxe des requêtes qui nous permet de spécifier les propriétés de notre configuration souhaitée.

À l'époque, un programme prototype mettant en œuvre cette approche a été développée. Il a été écrit en OCaml [79] et basé sur la bibliothèque de résolution de contraintes FaCiLe [14]. Ce programme a ensuite évolué en un outil beaucoup plus sophistiqué et complexe, appelé Zephyrus, dont je suis le développeur principal. Dans les chapitres 6, 7 et 8, nous parlons de l'ensemble de la théorie derrière Zephyrus, nous décrivons les détails concernant l'outil lui-même (c'est-à-dire sa mise en œuvre et l'utilisation), et nous présentons les résultats d'une validation expérimentale de son utilité dans plusieurs contextes. Bien que ces trois chapitres sont ordonnés d'une manière spécifique, cet ordre n'est censé que faciliter la lecture et la compréhension de ce document et n'indique pas de séquence temporelle (comme c'était le cas pour chapitres pour les chapitres précédents), en fait nous devrions plutôt les considérer comme simplement décrivant trois aspects de la même chose.

L'outil Zephyrus et le variant étendu du modèle Aeolus, appelé le modèle Zephyrus, ont été développés à cause de la nécessité d'adapter notre approche fondamentale (basée sur les contraintes) à des considérations pratiques. Grâce au fait que toute la démarche avait été déjà bien établie, nous avons pu l'ajuster en toute sécurité et rendre notre modèle plus expressif d'une manière contrôlée, comme nous avons compris quel genre d'éléments pourraient y être ajouté sans briser notre méthode de l'encodage en contraintes. Ainsi, nous avons étendu le modèle avec un nouveau trait important: une description des machines disponibles où les services peuvent être déployés, avec les repositoires de paquets installés sur ces machines, ainsi que les ressources fournies par chaque machine et consommées par les services. En plus du modèle amélioré, un langage de requêtes beaucoup plus riche a été proposé (permettant de spécifier des propriétés plus avancées concernant la solution désirée). Aucun changement radical ou difficile à notre approche de base n'a été nécessaire afin de l'adapter à ces modifications. Toute cette approche élargie est présentée en détail et prouvée correcte dans le rapport technique *Optimal Provisioning in the Cloud* [31] ainsi que décrit dans le chapitre 6 de cette thèse.

L'outil Zephyrus est un prototype de recherche, développé de façon incrémentale. Il a rapidement évolué côté à côté au modèle formel, il était constamment évalué dans divers contextes et adaptait en permanence aux différents besoins. D'un côté il est basé sur de solides fondements théoriques de modèle Zephyrus et nous le pouvons voir essentiellement comme une mise en œuvre directe de notre approche

formelle étendue. De l'autre côté c'est un outil qui fonctionne dans le monde réel, qui donc contient une énorme quantité de savoir-faire pratique et des solutions concrètes aux problèmes de bas niveau, comme:

- la lecture, le traitement et l'écriture de l'entrée et de la sortie structurés, en plusieurs formats (par exemple, la configuration du système distribué peut être sortie sous une forme graphique d'un diagramme de déploiement);
- prétraitement et post-traitement complexe, visant à augmenter l'efficacité de la résolution des problèmes à contraintes et d'améliorer ainsi les performances de Zephyrus (y compris le prétraitement avancée des métadonnées des paquets logiciels, effectué par l'outil coinst [25]);
- le soutien pour des divers solveurs de contraints en back-end, la simulation de l'optimisation multi objectif avec l'optimisation simple objectif, la mise en œuvre de la méthode portfolio, qui utilise plusieurs différentes solveurs de contraintes en parallèle;

et beaucoup d'autres. Ce visage pratique terre-à-terre de Zephyrus est présenté, avec tous les détails nécessaires, dans le chapitre 7.

L'approche de Aeolus à la synthèse de l'architecture et l'outil Zephyrus qui la met en œuvre ont été validés expérimentalement de plusieurs façons. Zephyrus a fait s'est prouvé efficace et utile non seulement dans une série de scénarios artificiels, mais aussi dans un environnement industriel d'une entreprise réelle (dans la société Kyriba), et en tant qu'un composant de la chaîne d'outils complète Aeolus, développée par la société Mandriva. Tous ces résultats sont décrits dans le chapitre 8, et la partie la plus essentielle d'entre eux a été publié dans l'article *Automated Synthesis and Deployment of Cloud Applications* [24] dans la conférence internationale ASE [2].

1.2 La structure du document

Ce document est divisé en plusieurs chapitres, chacun d'entre eux dédié à décrire un certain aspect du travail effectué et les résultats associés:

- Chapitre 2: *Introduction*

  Dans l'introduction, nous établissons une esquisse de base de notre sujet, nous expliquons comment le travail présenté ici s'inscrit dans la perspective plus large du projet Aeolus et nous fournissons un résumé du contenu de cette thèse.

- Chapitre 3: *Context*

  Dans ce chapitre, nous effectuons une analyse de l'état de l'art dans le domaine de la gestion des logiciels distribués. Nous présentons plusieurs technologies existantes, nous établissons les faits essentiels concernant leurs capacités et leurs limites, et nous les évaluons dans le contexte de projet Aeolus et de ses objectifs. De plus, nous introduisons un certain nombre des concepts généraux qui nous servent comme une base pour les chapitres suivants.

- Chapitre 4: *Aeolus model for distributed systems*

  Dans ce chapitre, nous présentons le modèle Aeolus pour les systèmes distribués, ainsi que les résultats théoriques associés. Nous commençons dans les sections 4.2 et 4.3 en introduisant étape par étape tous les éléments du modèle, en fournissant les intuitions derrière ces éléments, en discutent les décisions de conception liées et en établissant la correspondance entre le modèle lui-même et la réalité qu'il est censé modeler (c'est-à-dire le système distribué). Puis nous passons à formaliser l'ensemble du modèle Aeolus dans la section 4.4 et à parler de ses propriétés formelles dans la section 4.5 en définissant formellement plusieurs problèmes intéressants le concernant. Enfin, dans la section 4.6, nous montrons comment un véritable système distribué – une ferme WordPress – peut être représenté dans le modèle Aeolus, ce qui sert comme un exemple de l'expressivité du modèle.

[2] Automated Software Engineering

- Chapitre 5: *Synthesis of a component architecture*

  Dans ce chapitre, nous nous concentrons sur le *problème de la synthèse de l'architecture* et nous proposons une approche qui nous permet de le résoudre en pratique. Tout d'abord, dans la section 5.2, nous discutons minutieusement le problème lui-même. Nous introduisons un variant spécial sans états du modèle Aeolus, ainsi que quelques formalismes supplémentaires, qui nous permettent ensemble de spécifier précisément des instances du problème de la synthèse de l'architecture. Ensuite, dans la section 5.3, nous présentons notre approche, basée sur les contraintes, à la résolution de telles instances, en expliquant en détail chaque étape du processus. Tous les éléments de cette approche ont été formalisés et prouvés corrects.

- Chapitre 6: *Synthesis of a cloud-based architecture*

  Dans ce chapitre, nous nous appuyons sur nos résultats du chapitre précédent, en les étendent afin de rendre notre approche applicable aux environnements distribués réels. Nous commençons dans la section 6.2 en évaluant notre situation et indiquant clairement nos nouveaux objectifs. Puis, dans la section 6.3, nous présentons les principaux problèmes qui doivent être résolus pour atteindre ces objectifs et les plus importantes décisions de conception prises dans ce contexte. Par la suite, dans la section 6.4, nous montrons – dans la façon informelle et semi-formelle – comment ces décisions ont été mises en œuvre en étendant le modèle Aeolus sans états. Dans la section 6.5, nous formalisons complètement ce modèle Aeolus sans états *étendu* (aussi connu comme le modèle Zephyrus) et nous étendons également tous les formalismes définis dans le chapitre précédent. Enfin, dans la section 6.6, nous montrons comment nous avons ajusté notre entière approche basée sur les contraintes afin de l'adapter à la résolution des instances du *problème de la synthèse de l'architecture étendu*.

- Chapitre 7: *The Zephyrus tool*

  Dans ce chapitre, nous présentons et décrivons en détail l'outil Zephyrus, qui est un des résultats pratiques du projet Aeolus. L'outil est basé sur notre modèle Zephyrus théorique et il met en œuvre notre approche basée sur les contraintes à la synthèse de l'architecture.

  Nous introduisons Zephyrus dans la section 7.2 en présentant quelques informations de base sur l'outil lui-même et sur son flux de travail, et aussi nous mentionnons toutes les technologies externes qui sont utilisées dedans. Puis, dans la section 7.3, nous parlons des principes concernant l'efficacité de Zephyrus et nous présentons quelques idées de base à propos des possibles angles d'amélioration. Dans la section 7.4, nous suivons le workflow de Zephyrus étape par étape afin de montrer comment il fonctionne et comment il est utilisé en pratique (au moins sur le niveau basique). À la fin, dans la section 7.5, nous proposons et discutons en détail les techniques qui ont été utilisées, ou qui pourraient être utilisés, dans Zephyrus pour améliorer son efficacité.

- Chapitre 8: *Experimentation and validation*

  Dans ce chapitre, nous présentons les résultats de plusieurs tentatives de valider expérimentalement l'utilité et l'efficacité de Zephyrus, indépendamment, et ensuite dans le contexte de l'ensemble de l'approche Aeolus. Nous commençons, dans la section 8.2, en décrivant et en discutant plusieurs séries des tests basés sur des scénarios de référence fondés sur le cas d'usage ferme WordPress, que nous avons utilisés pour évaluer l'efficacité de Zephyrus dans la résolution des instances du problème de la synthèse de l'architecture de la taille et de la structure différentes. Puis, dans la section 8.3, nous présentons brièvement autres outils développés dans le cadre du projet Aeolus et nous montrons comment Zephyrus s'inscrit dans le contexte de la chaîne d'outils Aeolus. Enfin, dans la section 8.4, nous présentons comment Zephyrus a été utilisé avec succès dans un cas d'utilisation industrielle à Kyriba Corporation.

- Chapitre 9: *Related*

  Dans ce chapitre, nous tentons de mentionner systématiquement tous les articles scientifiques et toutes les technologies existantes qui ont inspiré notre travail, ou qui peuvent être considérés comme y liés dans une certaine façon.

- Chapitre 10: *Conclusion*

  Dans ce chapitre, nous récapitulons les effets de notre travail et nous essayons d'identifier au moins une partie des angles intéressants pour le développement ultérieur de nos résultats théoriques et pratiques.

CHAPTER 2

INTRODUCTION

The IT landscape underwent a radical change due to the availability of a first generation of technology offerings that go under the name of Cloud Computing. Even if this term is far from being precise, the general promise of the technology behind it is to offer cheaper and flexible IT resources (e.g. computing power, storage), which may be rented instead of acquired upfront, and quickly scaled up and down according to the end-user's needs.

Important challenges still need to be faced though. After navigating through the hype, typical of the first years of every new technology, users found out that the possibility of renting tens to thousands of virtual machines for a few weeks a year is just a first step: deploying, configuring, maintaining, and evolving the software applications and the systems on these virtual machines still requires months of work for highly qualified and expensive system engineers and administrators.

This thesis contributes to automating part of this work, by presenting an innovative approach to the synthesis of distributed component architectures that are particularly appropriate for the development and deployment of applications for the Cloud.

The work presented in this thesis has been performed in the framework of Aeolus, an ANR funded research and development project that involved the PPS Laboratory of University Paris Diderot, the I3S laboratory of the University of Nice-Sophia Antipolis, and the Inria Focus project. Aeolus was started in 2010 and ended in 2014, and was organised around three main research lines:

Task 1 Design an **abstract model** of a Cloud, which encompasses the software packages installed on every (virtual) machine, as well as the dynamic dependencies among software, machines, and services.

Task 2 Design a powerful **high-level reconfiguration request language** allowing to express sophisticated reconfiguration requests of software running on (virtual) machine pools.

Task 3 Develop specialised solver algorithms allowing **efficiently satisfy reconfiguration requests** by translating them into **low-level reconfiguration plans**, close to the actions implemented by common software deployment tools.

In the framework of this research program, our work is specifically focused on the modeling of distributed applications, the synthesis of application architectures from a high level specification language, through an encoding into numerical constraints, and the implementation of these ideas into a sophisticated and flexible tool, Zephyrus, that is now used as part of a complete toolchain developed by the Mandriva[1] company.

[1] http://www.mandriva.com/

2.1 Work done

My involvement in the work associated with the Aeolus project began at the first year of my PhD program by assisting in the research concerning the state of the art in existing cloud technologies. We were interested both in the industrial solutions employed in practice and in all the scientific studies potentially applicable in cloud configuration management and deployment automation. The aim was to understand what is exactly the current level of theoretical knowledge and practical know-how in different areas touching the Aeolus project's aims, discern the approaches and ideas most relevant in our particular context, that could inspire us or that we could build on, and determine where are the best opportunities for improvement.

The part of this research to which I have contributed the most was investigating some of the popular and widely used state of the art distributed software configuration management tools. I have performed an extensive study concerning a limited set of few such tools, attempting to achieve several objectives:

- analyse the existing working solutions and the typical use cases that they can deal with in order to understand better the whole distributed software configuration management problematic, as well as to recognise some issues naturally associated with it and the habitual methods of solving them;
- extract and compare the, implicit or explicit, *models* used in each of these tools to represent the managed distributed system, its configuration, and changes of that configuration;
- appraise these tools in context of potential integration in the Aeolus project as the low-level deployment back-end.

This study, which culminated in writing a technical report "Distributed System Administration Technologies, a State of the Art" [125], is described more in detail in chapter 3.

All these initial inquiries helped us considerably in establishing the *Aeolus model*: a simple yet expressive component model aiming to formally describe modern distributed systems and their reconfigurations from the administrative perspective, on the level of services operating in the system and relationships between them. This model was inspired by multiple ideas taken from various sources (see chapter 9 for details) and developed taking into account the knowledge we had about about common system administration requirements and the state of the art configuration management technologies.

At the heart of the Aeolus model there is a fundamental trade-off between expressivity and simplicity. It was designed to include the bare minimum necessary to model all the features which we considered essential: service dependencies and conflicts, life-cycle of services, certain non-functional constraints like redundancy and capacity, etc. It is thus expressive enough to capture the most important properties of real-life distributed systems, while in the same time it remains as simple and abstract as possible in order to be suitable for a formal study of the computational complexity of problems related with deployment automation. Above all it abstracts away from the details of the actual placement of services on the (virtual) machines.

The Aeolus formal model and its properties have been described in three published papers: *Towards a Formal Component Model for the Cloud* [36], *Component Reconfiguration in the Presence of Conflicts* [33] and *Aeolus: a Component Model for the Cloud* [34]. Although I have not contributed directly to these papers nor to the development of the model itself, in chapter 4 I present it and discuss in detail all its elements and properties, as it is fundamental to the approach described in the rest of this document.

Studying the computational complexity of software reconfigurations using the Aeolus model has yielded mostly quite limiting results. The formally defined *reconfiguration problem* (see section 4.5), which basically consists of searching for a way to reconfigure a given distributed system to make it reach a certain new desired configuration, has proven itself to be undecidable in the full Aeolus model and rather difficult to solve in all but the most simplified versions of the model. In order to proceed we have thus decided to apply the divide and conquer principle and split this problem in two almost orthogonal parts. Then we (here "we" refers to the Aeolus project team located in Paris) have focused on the *architecture synthesis* sub-problem (see section 5.2), namely searching for a new configuration for a given distributed system, which is valid, satisfies a certain request and is optimal according to provided criteria. Simultaneously another Aeolus project team (the INRIA Focus team) was attempting to tackle the complementary sub-problem of optimal deployment planning.

In chapter 5 I present in detail our constraint-based approach to the architecture synthesis problem, which I assisted in designing. It consists of three basic steps:

1. first we encode a given architecture synthesis problem instance into the form of a set of finite-domain integer constraints,
2. then we solve the constraint problem generated in such way using an external solver,
3. and finally we translate the constraint problem's solution back to the form of the original problem's solution.

The most challenging issue which we had to face when developing this approach was how to properly encode the redundancy and capacity requirements concerning services in form of integer-based constraints, and then how to establish in what manner these services should be bound together to assure correctness of the generated solution. This was achieved using a combination of two ingenious ideas: the so-called *binding unicity* constraints (see section 5.3.3) and a method called the *matching algorihm* (see section 5.3.4) used to generate correct bindings between instances of services.

All the pieces of this constraint-based approach have been formalized and proven correct. A special stateless variant of the Aeolus model was introduced to this end, together with a request syntax which lets us specify the properties of our desired configuration.

At the time, a prototype program implementing this approach has been developed. It was written in OCaml [79] and based on the FaCiLe constraint solving library [14]. This program later evolved into a much more sophisticated and complex tool called Zephyrus, of which I am the main developer. In chapters 6, 7, and 8 we discuss the whole theory behind Zephyrus, we describe the details concerning the tool itself (i.e. its implementation and usage), and we present the results of experimental validation of its usefulness in several contexts. Although these three chapters are arranged in a specific way, this order is only supposed to facilitate reading and understanding this document and does not indicate a temporal sequence (like the previous chapters did), in fact we should rather view them as simply describing three aspects of the same thing.

The Zephyrus tool and the related extended variant of the stateless Aeolus model, called the Zephyrus model, were developed because of the need to adapt our basic constraint-based approach to practical considerations. Thanks to the fact that the entire approach was already well established, we were able to tweak it quite safely and to make our model more expressive in a controlled way, as we understood what kind of elements could be safely added there without ruining our constraint encoding method. Thus we have extended the model with a significant new feature: a description of the available machines where services can be deployed, including package repositories installed on these machines, as well as computing resources provided by each machine and consumed by services. In addition to the enhanced model, a much richer request language was proposed (permitting to specify some more advanced properties concerning the desired solution). No radical or challenging changes to our basic approach were necessary in order to adapt it to these modifications. This whole extended approach is discussed in detail and proven correct in the technical report *Optimal Provisioning in the Cloud* [31] as well as described in chapter 6 of this thesis.

Zephyrus is a research prototype tool developed in a incremental manner. It evolved rapidly alongside the formal model, it was constantly evaluated in various contexts and adapted continuously to different needs. From one side it is based on strong theoretical foundations of Zephyrus model we can see it as basically a direct implementation of the associated extended formal constraint-based approach. From the other side it is a real-world tool which contains a huge amount of practical know-how and concrete solutions to low-level problems, like:

- reading, processing and outputting structured input and output in several formats (e.g. the distributed system configuration can be outputted in a graphical form of a deployment diagram);
- complex pre-processing and post-processing aiming at increasing solving efficiency and thus improving Zephyrus performances (including advanced package meta-data pre-processing performed by the coinst tool [25]);

- handling various solver back-ends, simulating multi-objective optimization with simple-objective optimization, implementing the portfolio solving method which uses several different constraint solvers in parallel;

and many many other. This practical down-to-earth face of Zephyrus is presented, together with all the necessary details, in chapter 7.

The Aeolus approach to the architecture synthesis and the Zephyrus tool which embodies it have been validated experimentally in multiple ways. Zephyrus has proven itself to be efficient and useful not only in a series of artificial benchmark scenarios, but also in real corporate industrial environment (in the Kyriba corporation), and as a component of the complete Aeolus tool-chain developed by the Mandriva company. All these results are described in chapter 8, and the most essential part of them has been published in the article *Automated Synthesis and Deployment of Cloud Applications* [24] in the ASE [2] international conference.

2.2 Document structure

This document is divided into several chapters, each of them focused on describing a certain aspect of the performed work and its results:

- Chapter 2: *Introduction*

  In the introduction we establish a basic outline of the topic, explain how the work presented here fits into the broader perspective of the Aeolus project and provide a summary of the contents of this thesis.

- Chapter 3: *Context*

  In this chapter we perform a survey of the state of the art in the field of distributed software management. We present several existing technologies, establish essential facts concerning their capabilities and limitations, and assess them in the context of the Aeolus project and its goals. Moreover, we introduce some basic background information which serve as base for the subsequent chapters.

- Chapter 4: *Aeolus model for distributed systems*

  In this chapter we present the Aeolus component model for distributed systems, as well as the associated theoretical results. We begin in sections 4.2 and 4.3 by introducing step by step all the elements of the model, providing the intuitions behind them, discussing related design decisions and establishing the correspondence between the model itself and the reality it is supposed to model (i.e. a distributed system). Then we move on to formalize the entire Aeolus model in section 4.4 and discuss its formal properties in section 4.5 by defining and investigating several interesting problems concerning it. Finally, in section 4.6 we show how a real distributed system – a WordPress farm – can be represented in the Aeolus model, which serves as an example of the model's expressivity.

- Chapter 5: *Synthesis of a component architecture*

  In this chapter we focus on the *architecture synthesis problem* and we propose an approach which lets us solve it in practice. First, in section 5.2 we discuss in depth the problem itself. We introduce a special tailored stateless variant of the Aeolus model, as well as some more formalisms, which together let us specify precisely instances of the architecture synthesis problem. Next, in section 5.3, we present our constraint-based approach to solving such instances, detailing each step of the process. All the elements of this approach have been formalized and proven correct.

- Chapter 6: *Synthesis of a cloud-based architecture*

  In this chapter we build on our results from the previous chapter, extending them in order to make our approach applicable to actual real-life distributed environments. We begin in section 6.2 by assessing our situation and clearly stating our new goals. Then, in section 6.3, we outline the main

[2] Automated Software Engineering

problems which need to be resolved to reach these goals and the fundamental design decisions taken to this end. Subsequently, in section 6.4, we show – informally and semi-formally – how these decisions have been implemented through extending the stateless Aeolus model. In section 6.5 we fully formalize this *extended* stateless Aeolus model (also known as the Zephyrus model) and we also extend accordingly all the formalisms defined in the previous chapter. Finally, in section 6.6, we show how we have adjusted our entire constraint-based approach in order to adapt it to solving instances of the *extended architecture synthesis problem*.

- Chapter 7: *The Zephyrus tool*

  In this chapter we present and describe in detail the Zephyrus tool, which is one of the practical outcomes of the Aeolus project. It is based on our theoretical Zephyrus model and it implements our constraint-based approach to architecture synthesis.

  We introduce Zephyrus in section 7.2 by presenting some basic information about the tool itself and its workflow, as well as mentioning all the technologies used in it. Then, in section 7.3, we discus some principles concerning Zephyrus efficiency and we preview some ideas related to improving it. In section 7.4 we follow the Zephyrus workflow step by step in order to show how Zephyrus works and how it is used in practice (at least on a basic level). At the end, in section 7.5, we propose and discuss in detail the techniques which have been used or could be used in Zephyrus in order to improve its efficiency.

- Chapter 8: *Experimentation and validation*

  In this chapter we present results of several attempts to validate experimentally usefulness and efficiency of the Zephyrus tool, both on its own and in the context of the whole Aeolus approach. We begin, in section 8.2, by describing and discussing several series of benchmarks based on the WordPress farm use case which we have used to assess the Zephyrus efficiency in solving architecture synthesis problem instances of different size and structure. Then, in section 8.3, we briefly introduce other tools developed in the scope of the Aeolus project and we show how Zephyrus fits in the Aeolus tool-chain context. Finally, in section 8.4 we present how Zephyrus was used successfully in a large industrial use case at Kyriba Corporation.

- Chapter 9: *Related work*

  In this chapter we attempt to mention systematically all the scientific papers and all the existing technologies which have inspired parts of the presented work or may be considered as related to it in some way.

- Chapter 10: *Conclusion*

  In this chapter we recapitulate the effects of our work and we try to identify at least some of the interesting angles for further development of our theoretical and practical results.

In order to facilitate comparisons between formal definitions concerning the three models introduced in this document, as well as our constraint-based approach to the architecture synthesis, table 2.1 points to the parts of chapters 4, 5 and 6 corresponding to the analogous parts of each model / approach version.

	Chapter 4	**Chapter 5**	**Chapter 6**
Title	Aeolus model for distributed systems	Synthesis of a component architecture	Synthesis of a cloud-based architecture
Model name	the ***Aeolus model***	the ***stateless Aeolus model***	the ***extended stateless Aeolus model*** or the ***Zephyrus model***
Model formalization and related definitions			
Model elements	component type def. 1, p. 48 configuration def. 2, p. 49	component type def. 10, p. 69 universe def. 11, p. 69 configuration def. 12, p. 70	component type def. 26, p. 120 package def. 27, p. 121 repository def. 28, p. 121 universe def. 29, p. 121 configuration def. 30, p. 122
Specification		subsection 5.2.4	subsection 6.5.2
Configuration validity	validity def. 3, p. 50	validity w.r.t. universe def. 13, p. 73 validity w.r.t. specification def. 14, p. 74	validity w.r.t. universe def. 34, p. 125 validity w.r.t. specification def. 35, p. 127
Optimization definitions		subsection 5.2.6	subsection 6.5.3
Problem definitions		problem and solution def. 16, p. 78 optimal solution def. 17, p. 78	problem and solution def. 36, p. 132 optimal solution def. 37, p. 132
Constraint-based approach to solving the architecture synthesis problem			
Constraint syntax and semantics		syntax table 5.1 semantics table 5.2	extended optimization criteria table 6.4
Encoding into constraints		subsection 5.3.3 validity w.r.t. universe table 5.3 validity w.r.t. specification table 5.5 full constraint problem def. 22, p. 92	subsection 6.6.2 validity w.r.t. universe table 6.6 validity w.r.t. specification table 6.7 full constraint problem def. 39, p. 140
Generating final configuration		subsection 5.3.4: - components p. 95, - bindings p. 96, final configuration def. 23, p. 101	subsection 6.6.3 - locations p. 142, - components p. 143, - bindings p. 144, final configuration def. 40, p. 144

Table 2.1: Selected corresponding sections and definitions presenting our three models.

CHAPTER 3

CONTEXT

3.1 Motivation

At the beginning of the Aeolus project, we set out to analyse and understand the current state of the related technology in the field of distributed software management, and to identify the best opportunities for improvement.

We have studied the state of the art in cloud technologies and distributed software management tools. A technical report on this topic was prepared, establishing essential facts concerning the capabilities and limitations of the existing technology in the context of the Aeolus project and its goals.

We synthesise here the findings of this report, which is an important starting point for the rest of this document, as it introduces the background information needed to understand several aspects of the work done in the Aeolus project.

My contribution

This chapter is a short summary of the results presented in the technical report "Distributed System Administration Technologies, a State of the Art" [125], that I wrote in 2012.

Considering the fast pace at which the technologies examined here evolve currently, some details present in the report may be outdated. However, most of its conclusions remain valid, as we are not aware of any spectacular and game-changing breakthroughs in this field having happened in the meantime.

3.2 Introduction

The term *distributed configuration management* refers to the problem of managing the configuration of software components on multiple hosts, which are usually working together in a computer network. In most cases this task is pursued by *system administrators*, who are responsible for planning and carrying out all the required modifications: installing, removing, upgrading and configuring software. Traditionally they perform these operations "by hand", accessing the machines either directly or through the network (e.g. using SSH [124]).

When the demand for computing resources increases, be it in the case of simple workstations or expensive web servers, the number of hosts to manage grows. With them, the number of software components and the number of their relationships and dependencies raises as well.

The resulting increasing complexity issue has been partially offset by the progress of technology. Constantly augmenting capabilities of hardware and adoption of new solutions, like virtualization and cloud

computing, make it easier and less expensive to create big networks of machines. But the work of a qualified system administrator is not as easily scaled up. Hence the popularity of solutions aimed to help system administrators to handle the resulting complexity.

A *distributed configuration management tool* is a piece of software, which helps a system administrator to:

- define, store and maintain reusable configuration artefacts,
- automate the process of enforcing configuration changes on multiple hosts at a time.

A basic improvised tool of this kind could be as simple as a shell script, that is used to execute the same administration command on a set of remote machines, which are reachable via SSH.

Modern distributed configuration management tools are far superior to this example and offer much more in terms of both functionalities and efficiency. Most of them feature some kind of a specialized abstract language used to describe configurations, offer a centralized and structured way to store configuration descriptions, and allow to easily define which parts of the global configuration are relevant for which groups of machines. Moreover, such configuration management tools take care of the process of deploying the desired configuration on the hosts without requiring per-machine human interaction.

Typically in these advanced tools all the configuration information is stored on the central server and distributed between hosts not by *pushing* it from server to hosts, but rather by so-called *pulling*. Each host queries the server periodically about its new desired configuration (and / or how to attain it) and then it applies required changes locally. This way the whole system is more scalable, as some part of work is shifted form the configuration server to the hosts.

An important concern associated with distributed configuration management is *service orchestration*. This broad term contains all the aspects of automated coordination and management of services dispersed between multiple hosts. As services running on different machines participate in implementing a single coherent service relevant for end-users, each service should be changed in accordance with others. For instance, a depended upon service should not be shut down (e.g. for upgrade purposes) before the services that depend on it, and vice-versa for its start-up (e.g. at the end of its upgrade). Proper coordination is not an easy task, as services not only depend on each other and cooperate in various ways, but also depend on a whole range of software components that should be locally installed on the machines where the services run. Modern distributed configuration management tools offer a varying degree of service orchestration capabilities.

3.3 Relevance of the study for the Aeolus project

One of the important axes of the Aeolus project was to develop a *formal model* of complex distributed component-based software systems that, as it happens with the so called "clouds", can change in size over time due to the addition and removal of new (virtual) hosts. The formal modelling of such systems was then intended to be used to automatically plan software reconfigurations starting from high-level request submitted by system administrators.

To achieve that goal, the formal modelling should not be separate from the reality of existing technologies. It would have indeed be pointless to create a very expressive formal model, if instances of it could not faithfully represent the reality of actual clouds, or local networks, for the purposes of planning software upgrades or other reconfigurations.

The purpose of this study was thus to review and compare the state of the art of technologies for distributed configuration management. In particular, we focused on those technologies that were applicable to deploy FOSS[1] components on UNIX-like machines, as they constituted the technological *milieu* of interest for Aeolus. The review and comparison was meant to be useful for Aeolus for several reasons:

- Aeolus aimed at developing real and useful technology. To that end we needed to start from what was really used by actual system administrators at the time and ensure that all the use cases, which

[1] Free and Open Source Software

the existing technologies had been developed to solve could be grasped by the formal models Aeolus was going to develop.

During this study we have also found many recurrent concepts that existed throughout all reviewed technologies. They have formed a sort of informal "ontology" that corresponded to the way of thinking of system administrators. Such knowledge was very helpful as guidance during the development of Aeolus formal models.

- We had anticipated that, during the lifetime of Aeolus we were going to need to test the tools, that were going to be developed as part of the project, on real data and compare the outcome of planners with what system administrators would have done "by hand". Knowing existing technology allowed us to evaluate the feasibility of obtaining test data as translation from languages supported by existing tools to Aeolus-specific languages that were developed by Aeolus partners.

- On the other end of the spectrum, we hoped that we could reuse existing technologies as low-level deployment tools. If that turned out to be feasible, we imagined assembling fruitful pipelines where Aeolus planners could be used to solve complicate reconfiguration headaches (e.g. major upgrades of a FOSS distribution on all the machines that constitute a private cloud), and existing distributed configuration management tools would be used only for the actual deployment of configuration changes.

3.4 Object of the survey

We have chosen some of the most popular distributed configuration management tools available on the FOSS market for UNIX-like machines, and we have studied their functionality and architectures. The work has been based mostly on the available documentation for the tools and on already existing examples of their use, like provided patterns and solutions. In certain cases we also applied some amount of practical experimentation.

The purpose of this research can then be summarized as:

- To better understand the problem of distributed configuration management, by analysing existent working solutions.

- To extract the *models* used, explicitly or implicitly, by each tool and grasp the similarities and differences between them.

- To spot interesting mechanisms, methods and solutions of sub-problems employed in the researched tools.

- To examine and appraise the tools in context of potential application in the *Aeolus Project* as the low-level deployment layer.

We have considered the following distributed configuration management technologies:

1. CFEngine3 (by *CFEngine*)
   Third version of one of the first true distributed configuration management tools. Flexible, versatile and robust, but lacking the focus and ease of use of its competitors. It works on a rather low level of abstraction. We can find many basic CFEngine3 ideas and solutions reused in other tools.

2. Puppet (by *Puppet Labs*) and MCollective
   A very popular tool. It models the configuration of managed hosts by introducing the concept of abstract resource layer. Also it features a high focus on declarativity. When extended with MCollective, it gains a broader perspective and some capabilities of service orchestration.

3. Chef (by *Opscode*)
   In a nutshell, Chef is Puppet reworked from another point of view, abandoning the principle of declarativity and giving more direct control to the administrator. Also, it introduces several interesting mechanisms permitting a certain level of service orchestration.

4. JUJU (by *Canonical*)
   An entirely different approach to the distributed configuration management, dedicated completely to cloud environments. In fact it is much more a service orchestration framework than configuration management tool. JUJU is highly focused on the services and their relations and neglects many aspects of configuration management which fall below this level of abstraction. Moreover, its desire to retain a simple and clean model makes it simplistic and restrained.

3.5 Methodology

We have discerned several important and interesting aspects concerning functionality and architecture of the distributed configuration management software. In order to examine the tools objectively and effectively, we have organized these aspects into five main axes of comparison:

Domain All the configuration management tools are, by definition, designed to manage the configuration of some software components. We define the whole area which can be controlled by a tool as its domain and all the different objects controlled by the tool (existing inside the domain) as the resources.

1. What can the tool control? Where are the borders of his domain?
2. What is the catalogue of available resources?
3. How precise control do we have over each resource's state and behaviour?
4. Do possibilities of defining new types of resources exist?
5. Can we express relations (like dependency or conflict) between the resources?
6. What notions of location of resources exist? Can we distinguish between the resources on different machines?

Language Every configuration management tool needs to allow the system administrator to express the desired state of the system and / or describe actions needed to attain it. All of the compared tools use some kind of language to either specify the desired configuration itself or to write configuration scripts (which are executed in order to arrive at the desired system configuration).

1. Is the tool using a special DSL (Domain Specific Language) or a generic programming language?
2. What is the expressivity of the language? Does it impose important constraints on what we can do and describe?
3. What is the level of declarativity of the language?
4. Is the language used to specify the resources themselves or rather to indicate actions performed on the resources?
5. How naturally can we express our ideas, concerning the system state or actions performed in the system, using the language? Is the language well adapted to its tasks?

Automation The very reason of using a distributed configuration management tool is to automatize tasks associated with deploying and maintaining the software components. Therefore the issue of practical benefits provided by a given tool seems to be crucial in order to assess its value.

1. What do we gain using the tool over what we can do by hand?
2. Are any advanced algorithms or solvers used?
3. What is the level of verification of correctness and feasibility of our decisions and actions?
4. Is the tool designed to handle the distributed configuration management and / or also orchestration of distributed services?

Architecture The structure of the tool itself is tied with the architecture (or possible architectures) of the environment, which that the tool is designed to manage.

1. What is the top-level schema of the tool's architecture?
2. What variety of the environment's structures can the tool handle?
3. Is the tool itself distributed or centralized? To what degree?
4. What is the configuration propagation method:
   - *push* (we actively enforce the desired configuration on every machine)
   - or *pull* (every machine asks for its desired configuration by itself)?

Platforms Different tools are designed to work with a different variety of platforms and operating systems.

1. Which platforms are supported by the tool?
2. What level of infrastructure's heterogeneity is easily handled?
3. How automatic is heterogeneity handling? How big part of these differences in configuration, which result purely form differences between the platforms, do we have to explicit?

The results of the survey are presented concisely in table 3.1.

3.6 Summary of capabilities and limitations

We have analysed in detail a comprehensive selection of tools that are widely used for *distributed configuration management*, by allowing to define, store and maintain reusable configuration artefacts, and offering means to automate the process of enforcing configuration changes on multiple hosts.

- First, all of these tools miss a high level automation layer: it is not possible to specify high level requests, and have the tool rely on an external, efficient constraint solver or planner to propose a reconfiguration plan, let alone an optimal one. The best that we can find is PUPPET's architecture, which is the only one built on top of a clear notion of a *resource*, and which provides mechanisms whose goal is to *converge* a system towards a particular configuration.
- We also notice, that none of these tools allows to declare incompatibilities among resources, while conflict relations do exist among the underlying components. This is a serious limitation of all models, which are not faithful to the reality, and thus may lead to surprising and inconsistent configurations.
- We also remark that none of these tools has a real global model of resources, which would allow to declare relationships between resources placed on different machines.
  - Either we find a flat model, describing only resources in scope of a single machine, like in CFENGINE3, PUPPET, and CHEF (which are the most used ones);
  - or we find a flat model describing only services in the cloud, like in JUJU, where each service is identified with a particular group of machines.

  MCOLLECTIVE does add a notion of collection of machines on top of PUPPET, but without a unified model.

  This is a serious limitation, as it is currently impossible to declare dependencies between services, software components and machines in a unified way, while this is an essential precondition for being able to express sophisticated reconfiguration requests.

	CFEngine3	Puppet	what MCollective adds to Puppet	Chef	juju
Model					
domain	resources (e.g. file, package, service)	resources (e.g. file, package, service)		resources (e.g. file, package, service)	services (e.g. mysql, wordpress)
relations	none	relations between resources: dependency, refresh		imperative triggers between actions on resources	relations between services: provides, requires
control mechanism	promises about resources' state and behavior	declarations of desired state and behavior of resources	equivalent to sequences of desired states on all machines	declarations of actions on resources	definitions of services and their possible relations
hierarchy	flat model	flat model	two levels: machines and their resources	flat model	flat model
Language					
kind	own DSL	own DSL		an extension of *Ruby*	YAML / any executable language
style	declarative	declarative		imperative	declarative / imperative
use	actions on resources	resources state		actions on resources	services' metadata / configuration scripts for deploying and managing services
expressivity	restrained (but quite high)	restrained		full	restrained / full
pertinence	cumbersome, too declarative	well adapted		well adapted	well-adapted / too generic and low-level
Automatization					
benefits	equivalent to configuration scripts execution framework	automatic converging of resources to the desired state	infrastructure wide control of resources, orchestrating complex reconfigurations	fine grain control of distributed configuration, some orchestration capabilities	deployment and orchestration of services, only very primitive resource management
configuration management	high	high		very high	low
orchestration	low	low	medium	medium	high
requests	no	no		no	no
solver	no	no		no	no
Architecture					
architecture	distributed	centralized		centralized	centralized
particularities	by default simulates centralized architecture			thin server, thick client	cloud only
Platforms					
supported	all*	all*		all*	Ubuntu Linux
handling heterogeneity	low	medium		medium	none

* "all" supported platforms means roughly all common operating systems: *nix, Windows and Mac.

Table 3.1: The comparison summary table.

- Finally, we notice that all these tools only allow to talk about a single configuration of a system, be it the one deployed now, or the one that should be deployed next. There is no explicit notion of reconfiguration (or reconfiguration plan).

3.7 Conclusion

It appears that none of these tools is able to perform the tasks of reconfiguration planning which are at the core of the Aeolus project. However, some of them can be seen as potential target deployment layers for reconfiguration plans produced by the solvers developed in the framework of the Aeolus project. We believe in particular that any of CFEngine3, Puppet, and Chef could serve this role.

CHAPTER 4

AEOLUS MODEL FOR DISTRIBUTED SYSTEMS

4.1 Motivation

The main area of interest of the Aeolus project was to tackle the scientific challenges related to the administration of modern distributed systems. Our principal aim was mastering the complexity of cloud-based software architectures by developing theory and tools for automatic deployment and reconfiguration of such systems.

In order to *master the complexity* of these problem we needed to develop both adequate *theory* and *tools*. In other words, we wanted both to grasp the deployment related problems well theoretically and be able to solve them in practice.

The analysis of the state of the art in the cloud technologies and distributed software management tools (presented in chapter 3) allowed us to conclude that, although many tools automating various practical tasks related to the administration of cloud-based distributed systems have been created, almost all were based on ad-hoc solutions and lacked strong theoretical foundation.

In these circumstances the concept to apply a more scientific approach to this field seemed necessary. By pursuing a more methodological study of modern distributed systems we expected to be able to understand and eventually surmount the limits of existing technology.

The first step on the road to properly analysing and understanding entities as complex as distributed software systems was to develop a formal model for a modern cloud-based distributed system.

Initially the main purpose of creating this particular model was to study the computational complexity of the deployment automation problem. However it is worth noting that what we ultimately established as the *Aeolus model* is not a purely theoretical construct: extensions and variations of this model are indeed employed in the (fully operative) deployment tools produced by various teams of the Aeolus project.

My contribution

This chapter presents the Aeolus component model for the cloud, and its main properties, based on material from the articles *Towards a Formal Component Model for the Cloud* [36], *Component Reconfiguration in the Presence of Conflicts* [33] and *Aeolus: a Component Model for the Cloud* [34].

I am not one of the authors of these articles and the theoretical results presented in this chapter are not my work. However, I put a great effort here to provide a coherent, detailed and motivated presentation of the model, with a step by step introduction, with intuitions and in-depth analysis of many details of the Aeolus model and the related theory. Notably, I attempted to address the question of the relationship

between the model and the distributed system that it is modelling for the sake of establishing a more thorough interpretation concerning this aspect of the Aeolus approach.

4.2 Design principles of the Aeolus model

Before presenting the actual Aeolus model, we discuss here the design decisions concerning the model.

4.2.1 Expected expressivity

As we wanted to create a model of distributed systems which are based on existing software components, obviously the essential part of the job was to model the software components themselves and their interactions. However, the concept of *component interactions*, as we interpret it in this context, should not be limited to simple collaboration relations between components. We also wanted to represent some non-functional requirements [57], which are an important concern in real-life modern distributed systems, especially redundancy and capacity constraints. Moreover, as well as being able to express dependencies between components, we also wanted to allow expressing conflicts (a lesson learned from studying the package management problem, where conflicts are abundant).

Our other important concern was modelling the dynamic aspect of components: their life-cycle. Software components, especially when considered from the system administration and service orchestration point of view, can be seen as having an internal state (for example *uninstalled*, *installed* and *running*), which can change during their life. Their state affects their properties and behaviour, altering the way they interact with other components (e.g. most services need to be up and running in order to be able to interact with other entities). Keeping track of the component internal state's evolution and understanding how it affects the component's characteristics and relations is essential to handle causal dependencies between components (e.g. which component needs to be installed first) during operations like deployment or reconfiguration of the distributed system.

To sum up, the first postulate of the Aeolus model design was to make it expressive enough to model:

- software components,
- interactions between components,
- dependencies and conflicts between components,
- non-functional requirements like replication and capacity constraints,
- component life-cycle.

Many of the proposed features rather than touching the software-engineering aspect (i.e. development) of component based systems, belong to the domain of the deployment policy and system administration (i.e. operations). This fully complies with Aeolus philosophy, as taking into account those facets of the problem is essential in order to advance towards mastering the complexity of modern distributed systems deployment and reconfiguration: basically, when we use the Aeolus model we voluntarily assume the system administration perspective on the distributed systems.

4.2.2 Need for simplicity

Now let us consider the other side of the coin and see what constraints and restrictions had to be imposed on the designed model. In other words, how far could we go in detailing our representation of the real world and where should we draw the line of abstraction.

First of all, we observe that the whole point of creating a model is to simplify the reality. We needed to abstract some part of the complexity of cloud-based distributed software systems, because the model would not be useful if it was as complex as the real object.

However, in our case we had even more specific reasons to keep the model as abstract and simple as possible. The Aeolus model was designed to explore the computational complexity of the problems related

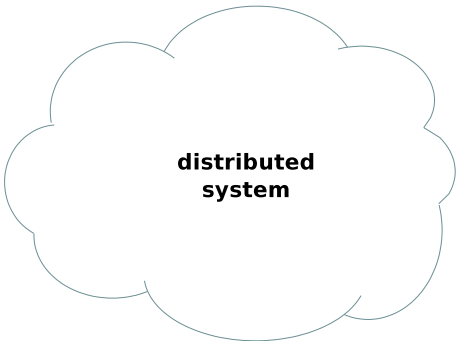


Figure 4.1: Our point of departure: the most abstract way possible to see a distributed system, where literally every detail has been abstracted away.

to the deployment and reconfiguration of distributed systems. Therefore, we needed a strict formal model, one which would be suitable for automatic treatment and reasoning. Basically this means that we aimed to push the line of abstraction as far as possible, while keeping it expressive enough to capture all the necessary features.

It is worth noting that many different variations of the Aeolus model were considered during the design phase, and even though the variant which eventually prevailed is indeed reasonably simple and abstract, it is still very expressive.

4.2.3 Conclusion

The decisions taken during designing the Aeolus model were based on the foundations presented above. The final Aeolus model is therefore a trade-off between expressivity and simplicity. On the one hand it tries to be as simple and abstract as possible, while on the other hand it attempts to capture many properties of the real-world systems which we considered essential.

4.3 Presentation of the Aeolus model

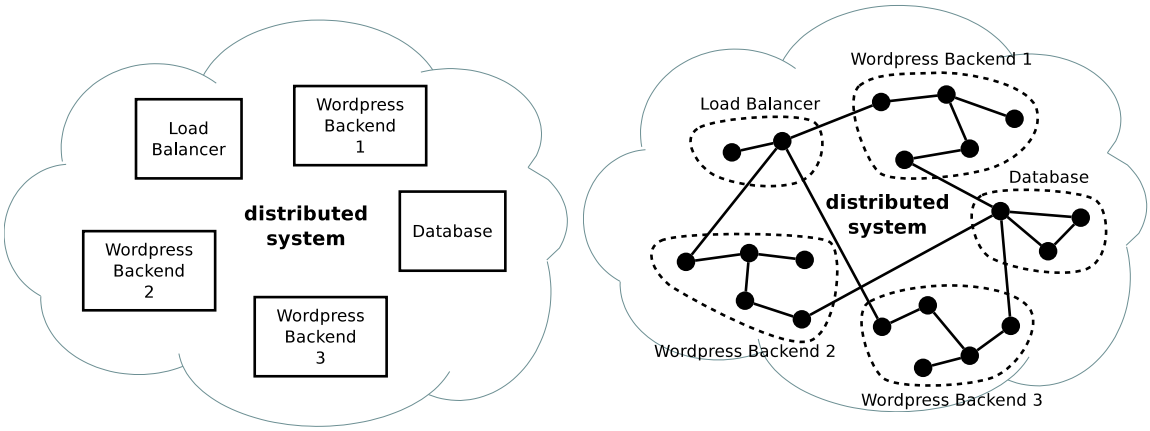
4.3.1 Introduction

To present the Aeolus model, we will first introduce all its elements in an informal way. We will explain what each feature of the model is supposed to represent in the real distributed system and how well its design matches the properties of the corresponding real-life entity. We also intend to explain the motivations of the design decisions concerning each part of the model in order to understand why they were made. At the same time we will systematically attempt to judge what are the possible advantages and shortcomings of various elements of the Aeolus model.

Presentation style

The model will be presented in a rather particular fashion, which will hopefully help us to achieve all these objectives. The principal idea here is that we begin with a completely abstract view of a distributed system, where we simply model it as a solid object with all the details about its internal structure abstracted away (like in figure 4.1), and then step by step we bring some aspects of its interior "out of the shadow" by including them in our model. This does not mean that all the other aspects are not taken into account at all. They simply stay "in the shadow": at the given point we do not include them in our current perspective (i.e. they remain completely abstract) even though we acknowledge their existence.

For example some of the concepts that will be introduced here are software components and relationships between them. At some point in the model presentation we will however only *see* the components and we will seem to ignore their relationships. This in fact means that in certain sense we will have already



(a) The Aeolus model software components.

(b) How we can imagine the distribution and relations of the software underneath the components.

Figure 4.2: With the difference to the previous perspective (figure 4.1), here we have "unabstracted" the different components which are part of the distributed system.

included the concept of software components in our model, but although we know that there are some relationships between them (and we certainly consider their impact when modelling the components), they are still not included in the model and stay "in the shadow", abstracted away for the time being (like in figure 4.2a).

Graphical notation

When presenting the model we will also introduce a simple graphical notation which allows us to depict all its elements in a highly readable and easily comprehensible visual fashion[1]. As the model will be presented incrementally, this graphical notation will be refined with each described fragment of the model.

4.3.2 Software components

The most essential concept of the Aeolus model is a software component, usually called simply a *component*. Basically a component in the Aeolus model is an abstract representation of a service operating in the distributed system.

Autonomy and unity

A *service* is a very broad term which can encompass pieces of software of various sizes and characteristics. Let us clarify what kinds of services we want to represent with our components.

There was a lot written on the topic of proper design of software components and component based systems, beginning in 1968 with Douglas McIlroy's *Mass Produced Software Components* [83]. For the purpose of this introduction we will not get too deeply into the related theory. We will just establish two simple informal guidelines that let us characterise the software that we want to see as a single component in the Aeolus model: *autonomy* and *unity*.

At the conceptual level a component is a whole and complete entity, which represents a coherent and complete part of a system. The underlying software should thus be reasonably unified and consolidated, e.g. all its parts should be deployed together on a single machine. It should also have at least a certain degree of autonomy, i.e. ability to function independently from the other components.

Unfortunately it is rather difficult to define clearly what exact level of autonomy do we have in mind here. For example a typical web application would be represented as an independent component in our

[1] Many parts of our graphical notation have been strongly inspired by UML [95].

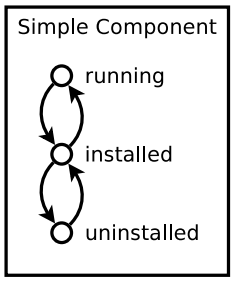


Figure 4.3: Example of a component with a simple state machine.

model, even if it requires a connection to a separate database component even to be configured correctly. Also if we wanted to model software which follows the master-slave architecture we would separate the master and slave components (which could be distributed on a different machines), even if obviously a slave component is useless on its own.

Although we can see some patterns emerging from these cases, we will not set any precise rules here and we will simply rely on intuitive judgements of what is autonomic enough in order to be modelled as a separate component in our understanding.

Component internal details

One important characteristic of the components in the Aeolus model is the fact, that what they represent is actually often composed of multiple smaller software elements. The basic idea behind the component concept is that a single component englobes a whole single *running service* and its configuration. Therefore it incorporates all what we could call the running parts of the service (all the code that is being executed, like daemons, client / server programs, etc.) and all the static parts as well (like software packages, configuration files, local storage, etc). In our model, we abstract the whole internal structure of the component, focusing only on how the component behaves as a whole and ignoring the low level details, which are not interesting from the distributed system administration perspective.

4.3.3 Component internal state

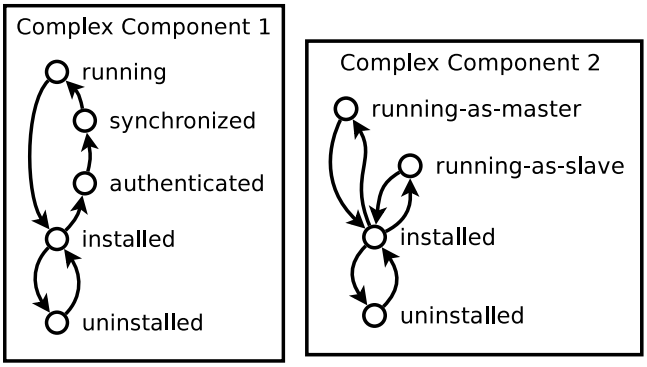
As we established before, one of the important features of software components which we wanted to include in the Aeolus model was their life-cycle. Most software components are not static from the system administration point of view, they have a dynamically changing internal state that affects their properties and behaviour.

Motivation

Intuitively in almost any service which we deploy in a distributed system we can distinguish at least two obvious internal states that we could call: `not running` and `running`. Of course this is a very simplified and high-level way of understanding the concept of an "internal state" of a piece of software. If we wanted to enter more deeply into modelling the details of functioning of any program, the representation of its internal state might get very complex. In the extreme case we can imagine that it could encompass everything down to the level of the full dump of the contents of its memory at any given moment of time, etc.

Getting into so much detail is definitely not our aim here. In the Aeolus model we rather want to abstract all the low-level details and stick to the high-level view of how the services operate. For us the important aspects of a software component's internal state are those which are relevant from the distributed system administration point of view.

Usually this means that we tend to care only about those internal state changes which really affect a given component's properties in a way that directly concerns other components and thus may introduce



(a) Example of a component with a state machine that has some additional intermediate stages.

(b) Example of a component with a state machine that has two different `running` states.

Figure 4.4: Example of two components with more elaborate state machines.

causal dependencies during system's reconfigurations. For example if a certain component needs a database connection in order to function properly and the database component is being shut down for maintenance reasons (which we can simply model as being switched to the `not-running` state) the aforementioned component will not work correctly any more and we will end up with a broken system. As we can see in this case, the order in which we switch on and off different components can be important when we are reconfiguring our distributed system. Hence we would like to keep track of this kind of internal state information in the model.

In practice the states that we include in the model resemble closely to the typical life-cycle of a real service: usually the set of possible states will contain at least elements like `uninstalled`, `installed` and `running` (or `initial`, `configured`, `running`). In some cases we will also add some more specific states in order to model more complex inter-component interactions requiring many stages or to allow more advanced types of service orchestration. Adding additional states is also natural if a certain component has many modes of functioning (e.g. `running-as-master` and `running-as-slave`).

There is one more aspect of the problem that we should keep in mind: the evolution of the internal state of a given real service is generally not random, it follows some strict rules. For example many services cannot go directly from being `uninstalled` to being `running`, they have to pass through the intermediary `installed` state (where they are ready to work, but have not been launched yet). Including this kind of restrictions in the model is necessary if we really want to reproduce the real services' behaviour.

State machine

After describing the background and explaining the reasons for key design decisions concerning the component internal state in Aeolus model, let us move on to the actual implementation of this feature.

In the Aeolus model each component is fitted with a finite state machine. Every state of that machine represents a certain internal state of the corresponding real service operating in the distributed system. Although in most cases these are simply the "big" steps of the service's life-cycle (e.g. `uninstalled`, `installed`, `running`, as depicted in figure 4.3), sometimes we may want to include also more subtle stages, like phases of a complex reconfiguration process (e.g. `authenticated`, `synchronized`) or multiple modes of functioning, as depicted in figures 4.4a and 4.4b.

At every given moment each component is in a single precise internal state, known as its *current state*. The current state can evolve, following the transitions available in the state machine. Each transition on the model level corresponds to some actions happening in the real system and leading to the change of the internal state of the matching service.

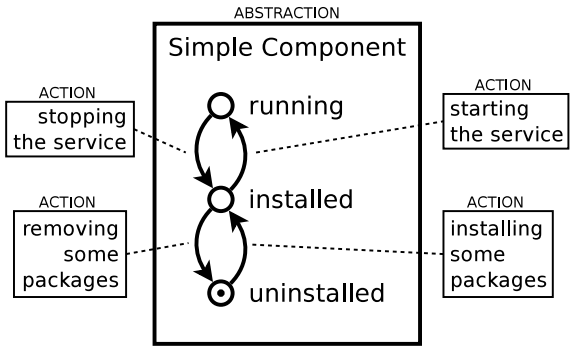


Figure 4.5: How the abstract concept of a state machine in a component maps to actions performed in the real system. (Note: The state with a dot is the current state.)

Mostly, what we call "actions" here refers to specific local administration tasks concerning the service in question. For example, passing from the `uninstalled` to `installed` state may be equivalent to installing an appropriate software package on a certain machine, and then passing from the `installed` to `running` state may simply correspond to launching the service (e.g. starting its daemon in the background). This kind of relation between the model-level state changes and the system-level actions is illustrated in figure 4.5.

We do not prohibit at all some degree of communication with other services happening during these actions. For instance, when a web application is switching its state from `installed` to `running` it may contact the database and query some information, then the database will respond, etc. The action is meant to be *local* from the administrative point of view: on the model level it should have effect only in the scope of the component which changes his state[2].

Abstraction level

As we can see, the way how the internal state of components and its evolution are represented on the Aeolus model level is highly simplified. Most of the complexity of what is really happening in the system is abstracted away: the internal state of a service is basically reduced to a limited set of discreet alternatives, we do not model at all the actual administrative actions performed nor the events which occur in software components themselves and we do not take into account the time they take. Instead we hide all that behind a high-level facade of a simple state machine.

We should note here, that there were a lot of model design choices possible here, between the enormous complexity of the real software system and the simplicity of a finite-state automaton. For an instance, we could just add variables to our state machine and we would get a one step more explicit model.

However, the simple state machine representation was chosen for a good reason. Thanks to it we keep the model as simple as possible, while leaving enough details to capture how the global component interdependencies are tied with the local state of each individual component. This fits very well the Aeolus model design philosophy of attaining the required expressivity (as defined in subsection 4.2.1) at the minimal possible complexity cost.

[2]The Aeolus model's purpose is modelling the administrative aspect of a distributed system. And the modelled system should not take administrative decisions nor perform administrative actions on itself. In the same way as a service does not use the package manager to install itself and then does not launch its own daemon, the components in the distributed system do not change their state on their own (nor the state of the other components).

A component's state change is supposed to model a conscious administrative action. And although there can be many things happening in the different parts of the distributed system because of that action, not to a degree that traverses the abstraction layer defined by the model. The effects of this action on other services operating in the system should simply not be relevant to the internal state of the components that represent them. Like in the example with the web application and a database: the web application cannot do anything that would make the running database shut down or update (of course unless as a result of an abnormal situation, e.g. a bug).

4.3.4 Collaboration of components

We have discussed in detail how individual software components and their internal states are represented in the Aeolus model. Let us see now how our model handles representing one of the most essential aspects of a distributed system: collaboration between different components.

Motivation

Component collaboration manifests itself in a multitude of various forms. We will use this term here very broadly, to refer in a uniform manner to all the different types of software component relationships which require components to work together on any level in order to attain a given functionality.

Often a collaboration will involve continuous direct communication between the collaborating components, like when an application is communicating with a database. But sometimes it may merely mean that the collaborating components are synchronized in some aspects of their configuration, which does not even need to entail any direct communication between them. For example, if we imagine that we have a backup load balancer whose configuration should match the configuration of our primary load balancer, it is the system administrator who will be responsible of assuring that this is the case and the involved software components will not even need to "know" about the collaboration.

In the Aeolus model we abstract over the fact that each of these relationships can have a very different nature underneath. What counts to us is how they behave from a higher-level point of view of the distributed system and to what kinds of dependencies between components they correspond.

Implementation: ports and bindings

Component collaboration is represented in the Aeolus model by *ports* and *bindings*.

Basic idea Components can be fitted with ports, which intuitively correspond to the functionalities that their real system counterparts provide or require. A binding between matching ports of different components represents an actual collaboration (in the context of a specific functionality) between the services corresponding to these components. In other words: ports on their own model a potential collaboration between components, whereas two ports bound together (i.e. linked with a binding) model an existing ongoing collaboration between components.

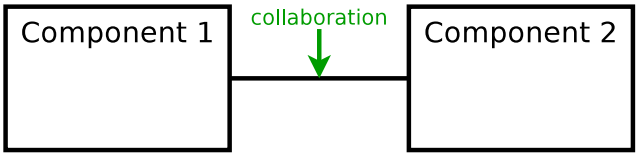
In figure 4.6 we can see how the Aeolus representation of collaboration with ports and bindings (the middle: figure 4.6b) tries to strike a proper compromise between a too simplistic abstraction (the top: figure 4.6a) and the too complex reality (the bottom: figure 4.6c).

Details Each port represents an aspect of collaboration or communication, usually a particular service that one component can provide and another can use. Every port has a name and all the ports which have the same name represent the same type of collaboration. Intuitively this is not a unique name which identifies each port, we should rather understand it in the same way as when we say "a USB port" or "a SSH port", etc.

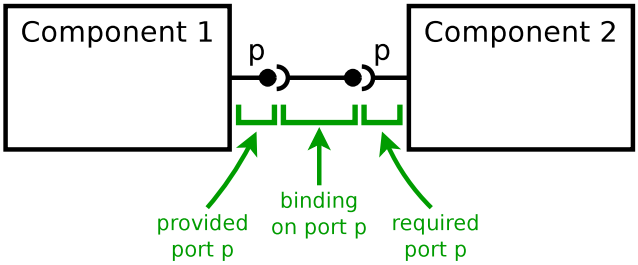
In the Aeolus model we use two complementary sorts of ports to represent the two sides of component collaboration: *require ports* and *provide ports*.

- A *require port* represents a functional dependency. It means that in order to work correctly a given component needs something that another component may provide.
- Conversely, a *provide port* represents fulfilling a functional dependency. It means that a given component provides a certain service that another component may need.

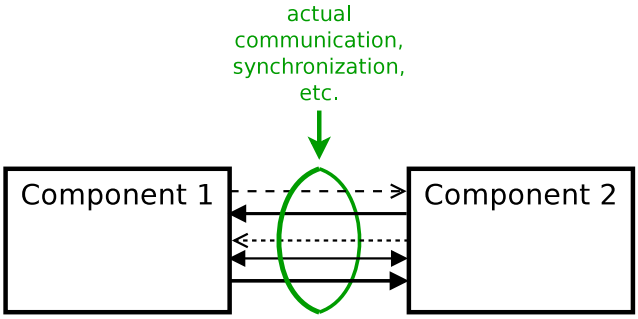
Bindings can exist only between require and provide ports which have the same name (thus representing the same aspect of collaboration), as obviously a given functional dependency can be satisfied only by providing exactly the required functionality.



(a) Completely abstract view of a collaboration. This is too high-level for the Aeolus model needs, as it gives us no idea how this collaboration works from the distributed system administration point of view.



(b) How the Aeolus model represents a single aspect of the collaboration with ports and bindings. This abstracts most of the details, yet lets us represent what is essential from the distributed system administration perspective.



(c) How we can imagine the collaboration in the real system. Basically a flow of events (e.g. messages sent and received by the two services) in time.

Figure 4.6: Different perspectives on two components collaborating. (In these pictures for the reasons of clarity we forget about the existence of the component state machines for the moment.)

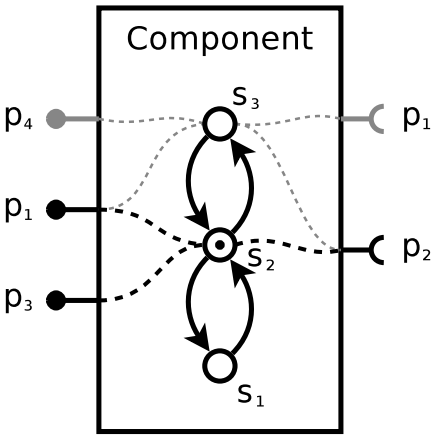


Figure 4.7: An example of a single component's ports and their activation. The state with a dot is the current state. We can see, that the port p_1 is both required and provided by this component.

A component can have many ports, though only a single provide port with a given name and only a single require port with a given name. It may simultaneously require and provide the same port, however it may not form a binding with itself and satisfy its own dependency.

Every port is *active* only when its component is in a certain defined set of internal states. Of course multiple ports can be simultaneously active in any given state (and nothing prevents having ports that are active in all states). We can see an example of a component with multiple inactive and active ports in figure 4.7.

Rules By coupling the port activation mechanism with constraints placed on state changes and bindings, we can model the fact that the component's behaviour and properties are shifting depending on its current internal state. The effects of being active or not depend on the type of the port:

- The main rule is that all the functional requirements (active require ports) of all the components must be satisfied (bound with active provide ports) at all times.

  More precisely: a component cannot be in a given state nor switch to it unless all his require ports which are active in that state are already bound with active provide ports.

- Therefore no binding with a currently active require port can be removed if this is the only binding of that require port with an active provide port (because there cannot be an active unsatisfied require port). In order to allow removing this binding, the concerned require port would have to be deactivated first (by switching the component to a state where this port is not active).

- Also, a component cannot deactivate a provide port which is currently bound to an active require port (i.e. if in the current state a certain provide port is active and bound, the component cannot switch to a state where this port is not active) if by doing this he is depriving the corresponding require port of its only binding with an active provide port (as it would be rendering this require port unsatisfied).

- A binding between two ports can exist even if the bound providing port and the bound requiring port are not active. In fact if both ports are inactive no additional conditions apply, such a binding can always be established or removed. Otherwise adding or removing a binding may be performed only if it does not contradict the stated rules.

Effect on state changes From these conditions come several observations about how the bindings behave in the model and how do they affect the possible state changes of components:

- First important effect that we can see is related to how the state changes are restricted by the non-active unbound require ports. In each component's state machine, all the transitions leading to states that activate any require ports which are currently unbound are obviously blocked. In order to unblock such transitions we need to bind these ports first (with active provide ports).
- An inverse effect is related to active bound provide ports. If in a given component's state certain provide ports are active and bound, then all the transitions to states which do not activate all these ports are blocked. In order to unblock such transitions we need to unbind these ports first (which is not possible without deactivating the bound require ports before).

Using these mechanisms we can model many simple and complex causal interdependencies between software components. We can see an example of how we could apply them to that end in figure 4.8.

Meaning of bindings

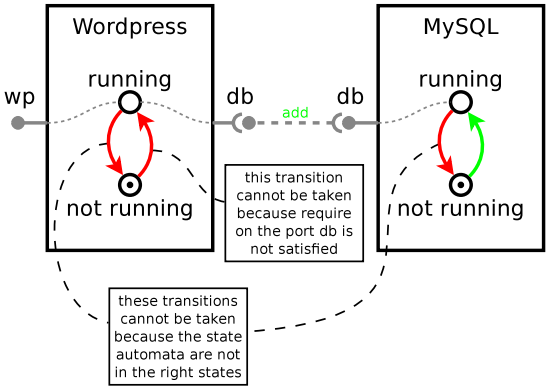
It is important to notice that, in the Aeolus representation of the collaboration between software components, the crucial part is definitely played by the bindings. Ports serve mostly as a way to model what potential bindings are available and how the presence or absence of bindings restricts the reconfiguration possibilities of our distributed system at a given moment. So let us forget for a moment about all the mechanisms related to the concept of ports and their activation in various internal states and let us focus more directly on the bindings themselves and what they represent.

Abstraction level As we already mentioned, a binding represents a collaboration or communication between two services operating in the distributed system. At the abstraction level of the Aeolus model we simply acknowledge the existence of this collaboration relationship between two given software components, without entering into any details about the exact nature of this relationship. Usually an established binding will correspond to an ongoing direct communication between the two considered entities, but it does not always have to be the case. Many other situations can be modelled using bindings: for example a single limited burst of communication between two components (like exchanging some configuration variables), that then keeps them synchronized in some way which does not require further sustained data exchange.

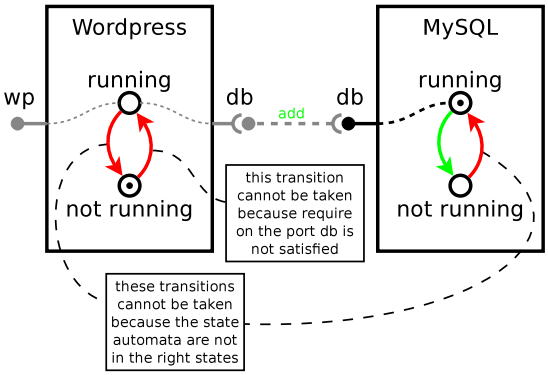
The understanding of what exact real system phenomena is hiding beneath a certain binding is of secondary importance at the model level. What we care most about is how this phenomena behaves in the system context from the administrative point of view: both how it is constrained by the system state (e.g. when it can be established or removed) and how does it constraint the system and its reconfigurations.

Life-cycle When we attempt to model the process of interaction between components by representing it in form of a binding, we try to attribute all the actions which are performed in the real distributed system to one of the three phases in the model:

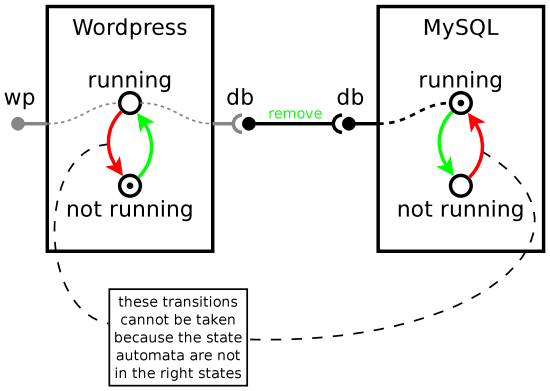
1. **Creation** of the binding, mostly corresponding to the phase of orchestrating the services to work together.
2. Stable **existence** of the established binding, mostly corresponding to the phase of the actual collaboration among services.



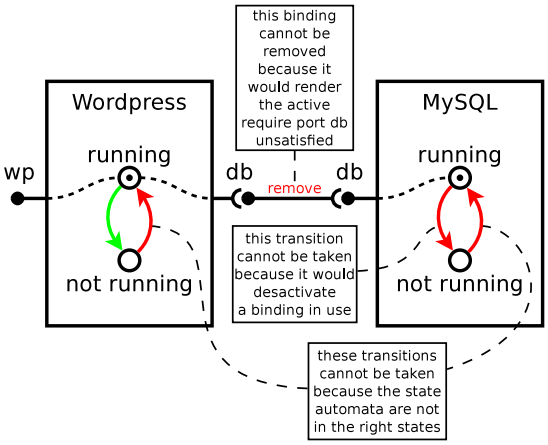
(a) Step 1: both components are not running, there is no binding between them.



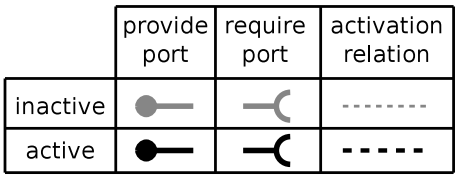
(b) Step 2: Only MySQL is running, there is still no binding.



(c) Step 3: Only MySQL is running, a binding has been established.



(d) Step 4: Both components are running, the binding is present.



(e) Legend

Figure 4.8: An example of switching states and creating bindings on two components. Green colour signifies actions which can be taken, red colour those that cannot be taken in the given situation. Both directions are equally correct: from Step 1 to Step 4 or the reverse.

3. **Removing** the binding, mostly corresponding to the phase of orchestrating the services to cease working together.

What is happening in each step can involve both some local actions performed on any of the two components as well as some communication between the components. Also these actions can belong both to the domain of "system administrator actions" (i.e. local reconfigurations or operations related to the service orchestration) and actions performed directly by the components themselves (e.g. database requests and responses).

This is how we can imagine that the binding life-cycle should correspond to the real process happening in the distributed system:

1. **Creation** of the binding.

   The system administrator prepares both services to work together (e.g. writes their respective IP addresses to local configuration files, prepares and synchronizes the authentication data like username and password, designates a network port to listen on for the server service and designates the same port to connect on for the client service, adjusts the firewalls on concerned machines appropriately) and then the services establish a connection (e.g. connect to one another, set up the communication, pass credentials and authenticate).

2. **Existence** of the binding.

   The services collaborate continuously (e.g. the application sends requests to the database and the database responds each time with results).

3. **Removing** the binding.

   The system administrator orchestrates both services to gently terminate their collaboration (e.g. commands the application to disconnect from the database).

As mentioned before, at the model level we do not want to see what exact actions are performed in the system nor what information is exactly passed between the components when. We stick to the highly abstract three-step view of the collaboration process and we leave the low-level details out of the model. Still, it is important to make sure that our model actually corresponds to the reality and that all the details which are left out at this level of abstraction actually fit into the schema.

Broader view Also we should keep in mind that the presented simple case of the component interaction is only an example. The way how the components collaborate may follow many different patterns. For example, as we mentioned before, often the services will perform all the actual communication in the *establish connection* step (a single burst of data exchange) and then they will do nothing when the binding is already there: the existence of the binding marks the fact that the two software components are coordinated or synchronized in some way, but does not entail any actual data transfer (for example a backup of certain services may work this way).

In fact we go even further in what we permit here: no system actions and no communication never really has to occur between the two components in the system. Our binding, as well as its creation and removal, can represent a process happening on an entirely different level, like for example the system administrator taking a note (e.g. "Y is the backup of X, I should switch to it if X fails"), which may affect the system in a certain indirect way in given circumstances (e.g. in this case it will influence the actions performed by the administrator in case of emergency).

The binding activation issue

There is one more thing to say about the correspondence between what happens with the bindings on the model level and the real actions performed in the distributed system.

Premise First, it is quite obvious, that at the time when a binding is created, its requiring port is never active. If it was active before creation of the binding, this would contradict the constraints that we have defined, as a component can not be in a given state unless all the requiring ports active in that state are bound. Second, as we know, any require port of a component can be bound at any state, there are no restrictions imposed on that.

This may led to situations, where a certain require port (e.g. database port) of a component is already bound when the component itself is still in a completely non-operational state (e.g. uninstalled), in which the actions related to the creation of the binding do not make any sense yet (e.g. the configuration files have not been created yet, so we cannot change the configuration variables defined in them). In this case the creation of the binding does not correspond to real actions performed instantly on the bound component, but it only hints that these actions will be performed sometime in the future, when they start to make sense.

Impact This means, that all that we have discussed concerning the correspondence between the phases of a binding's life (creation, existence and removal) and the processes happening in the distributed system does not always apply directly, but sometimes the events on the model level and system level are slightly decorrelated. We could, for example, imagine that the binding can be created at a certain moment, but until it is "activated" (i.e. its requiring port becomes active) all the actions corresponding to its creation are suspended and for the time it exists de facto only on the model level.

Consequences This issue may be seen as a slight shortcoming of the Aeolus model (as the timing of what happens in the model of the system does not match with the corresponding events in the modelled system itself), coming from the simplified way that the bindings were designed. Fortunately it does not cause too much trouble. We simply need to keep in mind, that this kind of additional retarded-binding-activation dependencies between bindings and states on the one side and actual actions performed in the system on the other side may exist, even if we do not perceive them at the model level. However, as long as the model manages to correctly represent the constraints concerning high-level causal dependencies in our distributed system, everything is fine.

4.3.5 Non-functional requirements

For now we have seen how ports and bindings are employed to model the functional requirements present in distributed systems. But in Aeolus model we wanted also to be able to express some so-called *non-functional* requirements.

Motivation

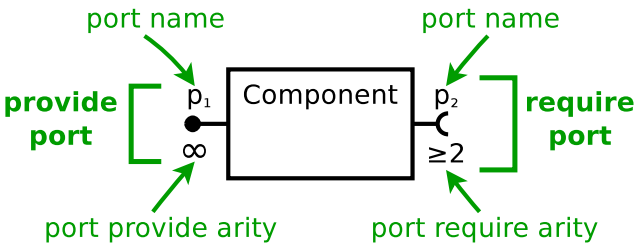
Non-functional requirements are often quite an important factor to take into account when designing and operating real distributed systems. Fulfilling non-functional requirements is not strictly essential to the system's correct functioning (hence the name), but it has to do with more sophisticated properties like quality of service, fault tolerance, redundancy, workload handling capacity, etc.

Implementation

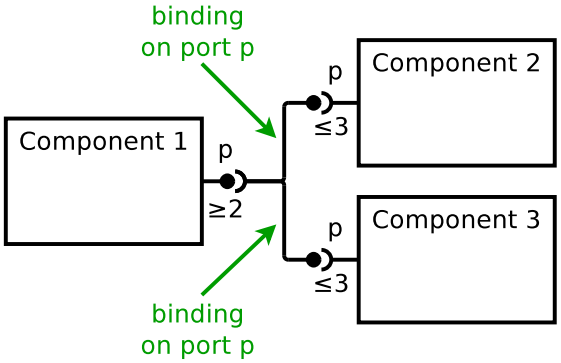
Aeolus model was not designed to let us express any imaginable non-functional requirement which could be applied to a distributed system and its components. This would be a very difficult thing to attain and it is far out of the scope that we have defined for the Aeolus project.

Nonetheless, Aeolus model contains features which make it possible to model some common non-functional requirements based on simple redundancy and capacity constraints (for graphical representation of the following see figure 4.9):

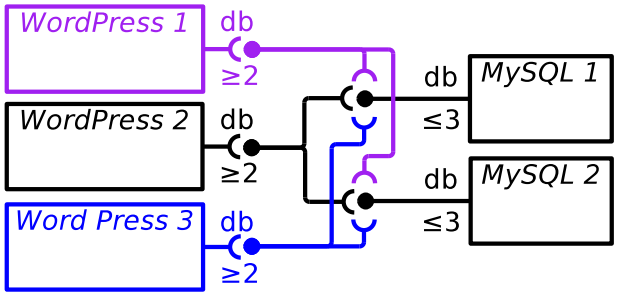
- We can express redundancy constraints in the Aeolus model by attaching a *require arity* to a require port in each state. Require arity is a positive natural number which specifies how many different



(a) Ports with arities.



(b) Correct bindings.



(c) A more concrete example of correct bindings. The provide ports are saturated here: we could not add any more bindings. (Different colours were used here solely to improve the readability of this web of intersecting bindings.)

Figure 4.9: Ports with arities and bindings in the Aeolus model. (For clarity we omit the component state machines in these diagrams.)

bindings (i.e. how many bindings with matching provide ports of different components) does this require port need to have in order to be considered satisfied. Its value can vary between states.

In a nutshell: require arity `n` means "must be bound to at least `n` **different** components".

- We can express capacity constraints in the Aeolus model by attaching a *provide arity* to a provide port in each state. Provide arity is a positive natural number which specifies what is the maximal number of bindings that this particular port can support at the same time. Its value can vary between states.

  In a nutshell: provide arity `n` means "can be bound to at most `n` different components".

Rules When establishing rules for ports and bindings in the previous section, we did not specify what exactly happens if a port is bound to multiple other ports in the same time. All the conditions imposed on ports were working well with single bindings and could be applied to situations where multiple bindings exist, as in fact the exact number of bindings was irrelevant to fulfilling functional requirements (i.e. a single binding is enough for that, additional ones do not change anything).

Now we will refine these conditions to take into account the require and provide arities and we will clarify all the ambiguities arising in the presence of multiple bindings:

- There are two main rules now:
  - All the functional requirements and non-functional redundancy requirements (corresponding to the active require ports and their require arities) of all the components must be satisfied at all times.

    The requirements corresponding to a given require port are considered satisfied if the port is bound to a number of different active provide ports (i.e. active provide ports which belong to different components), that is greater or equal to the port's require arity.

    Therefore a component cannot be in a given state nor switch to it unless all his require ports which are active in that state are already satisfied: bound with enough different active provide ports.
  - All the non-functional capacity requirements (corresponding to the active provide ports with non-infinite provide arities) of all the components must be satisfied at all times.

    A capacity requirement corresponding to a given provide port is considered satisfied if the port is not simultaneously bound to more active require ports than the value of his provide arity. If its provide arity is infinite, it can be bound to any number of require ports.

    Therefore a component cannot be in a given state nor switch to it if any of his provide ports which are active in that state would be then overused: bound to too many active require ports.
- No action can be taken (no component can switch its state and no binding can be created or removed) in a way that would result in an active require port being unsatisfied or an active provide port being overused:
  - A component cannot switch to a given state unless all his require ports that are active in that state are already satisfied.
  - Any binding with a currently active require port cannot be removed if it would make the port unsatisfied.
  - A component cannot switch to a given state if any if his provide ports that are active in that state would be overused.
  - A binding with a currently active provide port cannot be created if it would make the port overused.
- A binding between two ports can exist even if the bound providing port and the bound requiring port are not active. In fact if both ports are inactive no additional conditions apply, such a binding can always be established or removed. Otherwise adding or removing a binding may be performed only if it does not contradict the stated rules.

- A given require port and a given provide port can be bound with at most one binding (this rule applies even to bindings between non-active ports).

These new rules are mostly an obvious extension of the previous ones and they have similar consequences in the distributed system, only adjusted to take into account multiplicities of the provided and required bindings.

Binding unicity There is however one important detail that needs some further explanation: the last rule, introducing what we call the *binding unicity* principle.

In the Aeolus model we decided that existence of more than one binding on the same port between two given components makes no sense. As a binding does not stand for a certain amount of communication, two components are either bound or not, they cannot be less or more bound. Furthermore, considering the way we have defined require arity rules, a second binding with the same component would ether way be pointless: it would not add anything in terms of fulfilling the redundancy constraints (as it is the number of bindings with *different* components that counts).

Meaning of port arities

Let us see now what our redundancy and capacity constraints on the model level are supposed to mean in the context of the corresponding real distributed system.

Typical use The require arities are obviously useful to model all the one-requires-many relationships. We should note however, that these are not supposed to be functional requirements (i.e. one requires three to work correctly), but rather non-functional ones (i.e. one could work fine with one, but requires three because of high workload or fail safety reasons). So the require arities of our ports usually correspond not to real hard prerequisites, but more to our arbitrary deployment policy.

Typical utilizations of this mechanism may encompass simply implementing fail-safe solutions based on redundancy (e.g. although the component X requires to be bound with two different components, in reality it is using only one of them at any given time, but it will immediately switch to the other one if the current used one fails) and all variants of master-slave or frontend-backend patterns (e.g. the master component is dividing the work between multiple slaves, which permits it to handle bigger workloads efficiently).

Conversely the provide arities can be used to determine roughly what amount of workload a certain component should be able to support, given that we can estimate the amount of workload which each of the components using it will demand.

Deployment policy aspect The require and provide arities given to ports should not be automatically considered as inherent properties of their components, as they often belong more to the domain of the deployment policy of the whole system. Their exact values may depend strongly on the circumstances in which the components operate and usually only make sense in the context of a given distributed system.

Different systems based on similar components may be designed to satisfy different non-functional requirements. For example, the components used to model a toy small-scale version of a certain system can be almost exactly the same as the ones used to model a serious large-scale one (providing that the considered distributed architecture scales up easily), but their require and provide arities will generally change in order to reflect the bigger expected workload and more strict fail safety requirements.

In practice, a tool using the Aeolus model as an internal representation may well decide to maintain different *profiles* for the arities associated to some components, and create an instance of the model using a given profile only when a particular analysis must be performed.

Varying arities Another interesting feature of the require and provide arities as we defined them, is the fact that their values for a given port can vary from state to state. This gives quite a lot of flexibility and permits to model software components which can be configured in multiple different ways and their non-functional (and functional possibly too) requirements depend on their mode of functioning.

For example, we can imagine a service which is performing some computationally heavy tasks, that can operate either in a standalone mode, doing all the work locally, or in a parallel computation mode, where it takes the role of a master and distributes the computation between multiple slaves. Of course the workload which it can support is much larger in the second case. In order to model such a service we could fit the component that represents it with two separate `running` states corresponding to two modes of functioning:

- one that requires nothing and provides only small arity of the port related to the provided service (*running-standalone*),
- and another one that requires a connection with several slaves, but the provide arity associated with it is substantially bigger (*running-master*).

4.3.6 Conflicts between components

As we can see, the Aeolus model incorporates a whole set of mechanisms which let us represent various dependencies and collaboration relationships between the services operating in a distributed system. What complements this feature is capacity to also represent the conflicts which may arise between different parts of the system.

Motivation

In the software package management domain conflicts are abundant and solving problems related with avoiding conflicts is critical [25]. When designing the Aeolus model, we expected that the concept of conflicts and associated challenges were similarly essential to the distributed system configuration management. Therefore we have decided to include the possibility to express global conflicts between software components present in the distributed system.

Later we have found out that, even if local conflicts are in fact an important factor when designing and operating a distributed system, the global conflicts appear very rarely in such systems. Therefore the capacity to model global conflicts is in general not as useful as we believed it would be.

However, there is one important benefit of introducing it, as it allows to model a specific case of a global conflict that seems to be very common: services which should be unique in the system. If a software component needs to be unique in the system then it is in certain sense in conflict with any other component providing the same functionality as it. A typical case of this kind of service is a DHCP[3] server: we expect at most one to be present in our entire network. We call this kind of components *singletons*.

Implementation

Our implementation of component conflicts is based on the port mechanism, described in depth in the previous section. We simply introduce a new type of ports in addition to require and provide ones: *conflict ports*.

Intuitively a conflict port with a given name is in conflict with all the provide ports bearing the same name (thus usually representing a particular provided service). There is however one exception: this does not apply in the scope of a single component. Conflict ports on a certain component are not considered to be in conflict with ports provided by the same component. Self-conflicts are thus not possible.

In the context of the component internal states, conflict ports work similarly to the two other types of ports: every component can have many conflict ports (but only one with any given name) and each of them is only active exactly in a defined set of internal states. There is no arity associated to conflict ports.

Rules The conditions that conflict ports impose on components and state transitions complement those introduced by provide and require ports, and bindings between them:

- If a conflict port is active, then there cannot be any active provide ports bearing the same name present in the system. The only exception is the potential active provide port which belongs to the same component as the active conflict port. There is no conflict in this case.

[3] Dynamic Host Configuration Protocol [37]

- Thus a component cannot switch to a given state if it activates a conflict port that would be in conflict with any active provide ports already present in the system.
- Conversely a component cannot switch to a given state if it activates a provide port that would be in conflict with any active conflict ports already present in the system.

These rules are quite straightforward and intuitive, as basically they say, that there should be no conflict present in our system and we should not do anything that would introduce a conflict.

The exception concerning conflicts in scope of a single component gives us the possibility to easily model singleton services. We just have to make sure that all components in a certain group simultaneously provide a given service and are in conflict with it. Consequently at most one of these components (in a state that activates both of these ports) will be able to be present in the system at any given time.

Meaning of conflicts

Unfortunately it is not usually intuitively clear what do conflicts mean and when do they occur. Typically services operating in a distributed system are not "close enough" to cause any global incompatibility problems and frankly we did not really succeed to find any ordinary situations when putting two different specific software components together (without even making them interact) would break the system.

Singletons However, as mentioned before, there is one notable special case, when this happens commonly: the singleton components. There are some services that need to be unique in the scope of the whole system in order to work correctly (the most common example was already indicated: a DHCP server).

Moreover it happens quite often that, theoretically we could have many instances of a certain service present in the system, but in most practical cases we prefer to limit it to only one central instance and thus we model this service as a singleton component (for example, when considering DNS[4] servers, we may want to have only one master name server for our domain, even though it is technically possible to have many masters if we select one of them as the primary master). But we should bear in mind that, this kind of use of conflict ports should rather be regarded as encoding a specific deployment policy, not an actual conflict between two particular software components (or two copies of the same component).

Broader use Of course, as most of the other features of the Aeolus model, this one is also open to more creative use in order to encode some additional properties that, we want our system to satisfy. We can just add conflict ports to certain states of certain components to enforce or prohibit artificially specific possibilities of the distributed system's evolution. For example, if we have some kind of a principal component which should be completely set up before everything else, we could fit it with conflict ports conflicting with all the other components and make those ports active in all but its final configured state.

4.3.7 Configuration

We have covered almost all the important basic features of the Aeolus model. Let us take some time to summarize it and put together:

- We have components representing the services in the distributed system.
- Components possess internal state machines that model the changes of the proprieties and behaviour of their corresponding real-life services during their life-cycle.
- Components have ports that let them create bindings with appropriate ports of other components. Bindings represent various types of collaboration between services.
- Mechanism of port activation together with the restrictions on state machine transitions let us encode the causal dependencies between components which need to be taken into account during the reconfigurations of the system.

[4]Domain Name System [87]

- Ports can be fitted with require and provide arities which let us introduce some elementary non-functional requirements based on simple redundancy and capacity constraints.
- Finally components can also have conflict ports which represent global conflicts in the distributed system and are particularly useful to represent singleton services.

We know now more or less how all these elements fit together. We understand intuitively how do they correspond to entities present in the distributed system and how what happens in the model corresponds to the actions performed in the reality. We bear in mind, that in some cases our simplified representation of the real system can be slightly distorted, but it is all right as long as we stay coherent in the high-level administrative view of things. Also we realize that all these concepts can sometimes be used against their initial purpose in order to encode some specific properties that we would like to enforce on the system.

We will now proceed to complete this picture and clarify it with other concepts based on the already presented ones. Note, that this will not be yet a formal definition of the model, only a small step towards it.

Configuration definition

Let us introduce a concept bringing all the mentioned elements together: the *configuration*. A configuration is a set of components and a set of bindings between them.

Of course in the notion of a component here is already integrated its internal state machine, with ports and their activation relation. Moreover every component in a configuration has a single state selected as their current state.

We can say that what we call a configuration is a snapshot of the model of our distributed system taken at a certain moment of time. Each component is in a particular current state, the bindings which are there have been already established fully, everything is static.

Thanks to the concept of configuration we can attempt now to define more clearly two important ideas that have been appearing constantly when we were describing all the elements of the Aeolus model: the configuration validity and the system reconfigurations.

Configuration validity

The idea of determining either a configuration is *valid* or *invalid* incorporates many of the rules and constraints concerning various elements of the configuration that we have mentioned before. More specifically, as we have defined the configuration as a completely static object, the configuration validation will focus on static conditions which must be respected by our system in every certain fixed point of time. We will follow on to the dynamic properties later, when we talk about the system reconfigurations.

When we say, that a configuration is valid, we need to be sure that two groups of constraints are satisfied:

- The system is working correctly (i.e. the functional requirements are fulfilled).
- And all the non-functional requirements are satisfied (i.e. the redundancy and capacity constraints).

Structure But first of all, in order to even start considering if it is valid or not, the configuration must be technically sound and coherent. Certain pseudo-configurations are not really configurations by the Aeolus model definition. For example if in a given configuration exists a binding between two provide ports, then the object we are looking at is in fact simply not an Aeolus model configuration. Therefore, let us clarify what is a correct configuration's structure:

- A configuration consists of a finite set of components and a finite set of bindings.
- Each component in a configuration has a unique name.
- Each component in a configuration has an associated finite state machine with states and transitions between them. This machine is in a certain (single) current state.

- Each component in a configuration has associated sets of require, provide and conflict ports. These sets can be empty.

  Every port has a name which must be unique in the scope of its type in a given component (e.g. one component cannot have two provide ports called `database`, but can very well have one provide port and one require port with that name).

- Each port of a component has a set of states in which it is active (these must be the states coming from the state machine of this particular component). This set can be empty.

- Every biding is defined by a port name and two components: the requiring component and the providing component. The requiring component needs to have a require port with a matching name and the providing component needs to have a provide port with a matching name.

- As the configuration contains a set (and not a multi-set!) of bindings and each binding is defined exactly by three things (a port name, the requiring component and the providing component), there can exist at most two bindings on the same port between two components: one in each sense. Therefore there cannot exist more than one binding like that going in the same sense (i.e. with the same requiring and providing components).

  This is the already mentioned *binding unity* principle, now built directly into the configuration structure.

Functional requirements As we can see, this is not yet a formal definition of a configuration, but we are approaching it more and more. Let us now move on to a second set of conditions, those which correspond to functional requirements that the system has to satisfy in order to be deemed as working correctly. Enumerating them is quite simple:

- All the functional requirements must be satisfied.

  For every require port active in the configuration (i.e. a require port active in the current state of its component) there must exist a following binding in the configuration:

  - Binding is on this port.
  - With the component to whom belongs this require port being the requiring component of the binding.
  - With any other component (may be also the same component), that has a currently active provide port with a matching name, being the providing component of the binding.

- There can be no conflicts.

  If there is any active conflict port in the configuration (i.e. a conflict port active in the current state of its component) then there cannot be any active provide ports with a matching name present in the configuration, unless each of these provide ports belongs to the same component as their only corresponding conflict port (i.e. one component simultaneously providing a port and being in conflict with it does not create a conflict, but two of this kind of components do create a conflict, because the conflict port on each component is in conflict with the provide port of the other component).

Non-functional requirements And a simple extension of the first rule (the one dealing with the functional requirements), can be used to define the constraints which correspond to the non-functional requirements of the system:

- Requires must be satisfied.

  For every require port active in the configuration (i.e. a require port active in the current state of its component) the number of bindings present in the configuration and fulfilling the following conditions must be **greater or equal** to the require arity of that port:

  - The binding is on this port.
  - With the component to whom belongs this require port being the requiring component of the binding.
  - With any other component (may be also the same component), that has a currently active provide port with a matching name, being the providing component of the binding.

- Provides cannot be overused.

  For every active provide port in the configuration, the number of bindings present in the configuration and fulfilling the following conditions must be **lesser or equal** to the provide arity of that port:

  - The binding is on this port.
  - With the component to whom belongs this provide port being the providing component of the binding.
  - With any other component (may be also the same component), that has a currently active require port with a matching name, being the requiring component of the binding.

Together The three presented groups of constraints together define the structure of a configuration (the first group) and the notion of its validity (the second and third group).

One notable thing to notice is that the non-functional requirements constraint is in fact an extended version of the one responsible for fulfilling the functional requirements. If we simply set all the require arities in the configuration to one and all the provide arities to infinity, then those two rules are equivalent.

We should note, that this is an interesting simplified variant of the Aeolus model (i.e. one not including the non-functional requirements feature), which will be referred to later on when we talk about the model's expressivity and properties.

Binding unicity Another point of interest is the question of the binding unicity principle, and more generally: at what point do we decide if something is an invalid configuration or is not a configuration at all.

Here we assume, that the constraint of unicity of the bindings is woven directly into the very fabric of the configuration structure definition: an object which has two or more identical bindings is simply not an Aeolus model configuration. We acknowledge however, that this is purely a design decision and the only reason of putting the boundary between configurations and pseudo-configurations here is to make the definitions simpler. As the bindings which belong to a given configuration are a set (and not a multiset), and the only thing that differentiates one binding from another are its three basic properties (the port, the requiring component, the providing component), it is naturally impossible to have a double binding because it is impossible to put two identical objects in one set. We could have decided otherwise and either give bindings unique names or simply change the set of bindings to a multiset, thus moving the binding unicity principle from the domain of the configuration structure rules to the configuration validity rules.

4.3.8 Universe

Before we move on to clarifying the topic of the distributed system's reconfigurations, let us introduce a completely new concept, which is supposed to put some order in our unregulated repertoire of software components.

Motivation

Until now all the components we were talking about were considered individually, as the Aeolus model counterparts of specific instances of services operating in the modelled distributed system. But one of the essential characteristics of the typical modern distributed systems which we wanted to capture is the fact that they are built by assembling and connecting instances of pre-existing software components. So on the model level it should be clear, that the components which are present in our configurations are not some

independent pieces of software, each one tailored for its specific purpose, but that they all come from a certain well-defined collection of already existing elements.

This is where we introduce a new concept: the *universe*. A universe intuitively represents the assortment of building blocks which are used to construct a given distributed system. It complements the configuration, describing the available types of components that can exist in the configurations based on it.

Component types

The key ingredient of a universe are *component types*. A component type has almost exactly the same features as a component, but instead of representing a service actually operating in the distributed system it represents a service that potentially could be deployed in that system.

We may see it intuitively as a as kind of a blueprint for components (i.e. the components in the configuration are based on the component types available in the universe) or as something like an object programming class (i.e. the components in the configuration are instances of the available classes defined by the component types in the universe).

In spite of being a fundamentally different kind of entity, from the structural point of view a component type has nearly exactly the same characteristics as a component. It contains a state machine and it can possess require, provide and conflict ports, that stay in a certain relation with states of the machine (defining which ports are active in which states), with all these elements structured exactly like in an actual component.

However, there is one crucial difference: as component types are only blueprints for components, they are not meant to exist as such in the distributed system, thus they cannot change states or form bindings, they are completely static. And the structural detail which reflects this difference is the fact, that their state machines do not have a current state, but instead they have a single *initial state*. When a component type is instantiated in a configuration in form of a component, the newly created component's current state is the initial state of its corresponding component type.

Relationship with configuration validity

A universe is simply a set of component types, each with a unique name. It is used in the Aeolus model as a complement of a configuration: the repertoire of components which can exist in that configuration is restrained to the instances of component types available in the associated universe.

This introduces a new aspect to our concept of validity: configuration can be valid or invalid with respect to a certain universe. The rule to establish if a configuration is valid with respect to a given universe is simple:

- Every component present in the configuration must be an instance of a component type which exists in the universe. This means that their state machine, their ports and the activation relation (between the ports and the internal states) must be exactly the same.

Obviously there is no restriction on the current state of a component in the configuration, as it might have already changed from the initial one defined in the corresponding component type.

Shift of perspective

There is one interesting related observation, which should be mentioned here. There are in fact two possible perspectives that we can take on the relation between the concepts of the configuration and the universe:

1. Here we have defined a configuration from the beginning as something separate from the universe. We have said that if we want a certain configuration to be valid with respect to a given universe, each component from the configuration must have a corresponding component type in that universe. The universe is thus only a secondary entity and a configuration may or may not conform to constrains imposed by a certain universe.

2. However we could approach this situation from a completely contrary point of view and decide to put the universe in the centre of the model, by defining the configuration as in fact directly based on its corresponding universe. In this version of the model components in the configuration would not really have features on their own (which should match the features of their component type), but all they would have is a name, a current state and a type, which defines their characteristics directly by referring to a certain component type. In this case every configuration would necessarily need to have an associated universe, otherwise we could not describe its components at all.

This may seem as a purely formal difference, as in the end both cases behave in a similar way and have similar properties. Indeed in the Aeolus project sometimes we use one representation and sometimes the other, depending on what is more practical in a given case.

However (as we will see later), in most situations we tend to prefer the second approach (i.e. the one where each configuration is always compulsorily based on a certain universe and each component is based on a specific component type from this universe), because it is more clearly grouping components in homogeneous classes, which simplifies tremendously our reasoning. This solution also makes our model closer to the idea of distributed systems based obligatorily on pre-existing components.

Nevertheless, we should notice that the first representation (i.e. the one we were sticking to so far, where configuration is an independent entity) has some merits too, as it is in fact slightly superior to the second one in some aspects concerning expressivity:

- When defined this way (independently from the universe), every configuration can be valid (or not) with respect to many different universes. As in this version of the model we have not introduced any obligatory reference from the component to its type (we have only said, that there must exist a component type that matches each component), we are more flexible in changing the universe while keeping the same configuration.
- Moreover this actually permits us to describe existing configurations invalid with respect to a certain universe, which is a useful feature (e.g. it lets us pose questions like: how can we make this configuration valid?).

Universe validity

One last thing to note here: there is no such thing as a validity of the universe. There are no bindings in a universe, so as long as the structure of the universe is well-formed (it is a set of correctly described component types with unique names), it is automatically valid. If not, then it is simply not a Aeolus model universe.

4.3.9 Reconfigurations

Although describing formally (for the moment semi-formally) a distributed system is already interesting for us, the ultimate goal of the Aeolus model was to not only focus on the static aspect of such systems, but also be able to model their dynamic aspect. We will systematically call the whole process of transforming the distributed system from one state to another a *reconfiguration* or a *reconfiguration plan*.

Typical reconfigurations

There are three fundamental types of reconfigurations which we usually discern, corresponding to three elementary phases in the life of a distributed system. Though all three are essentially similar in theory, in practice we often approach them differently in order to profit from the regularities and specific properties each of them possesses:

Initial deployment This is the first reconfiguration which every distributed system has to undergo, passing from an empty environment to a fully configured, working system providing certain functionalities. Usually we start with nothing (a number of clean machines or simply a public or private cloud) and we have some kind of a description of the final system that should be implemented. Then we install and set up

components in the right order and we orchestrate them to work together in order to attain that final desired state.

The main regularity particular to the initial deployment is that the general direction of the reconfiguration is intuitively simple and constant: up. We are building the system: we install components, configure them, make them running, establish connections between them, etc. Very rarely any destructive action or even change of existing configuration is required. We simply add all the elements of the puzzle in the right order until we attain the goal.

The other specificity of this phase of the system's life is that we usually do not expect the whole to be operative and functional until the very end of the deployment process. We can safely assume that during the entire time of the initial reconfiguration there are no clients, users or external applications actively utilizing the system and depending on it to work. We can imagine, that nobody from the outside will access it until we decide that the set-up is over and we give a green light (e.g. open the ports on the external firewall).

Update After the system has been deployed, it has to be maintained. This entails performing changes on an already working system. In some situations it means, that we are required to keep it actively running and providing services during the whole time of the reconfiguration. In others we are allowed to have some downtime, when the system is not expected to be fully operational.

There are a few typical reasons of performing changes on a system that has been already set up. One of the most common reasons are software updates. Software, and especially highly popular open-source software, tends to evolve quickly: bugs are fixed, performances are improved, new features are introduced. In order to keep our system safe and up to date, we have to patch its elements regularly. Even if we are pretty conservative in this matter and prefer stability than supporting new features, we have to at least apply the most important security updates or we risk making it vulnerable to attacks.

Other reason of reconfiguring a running system is to scale it up or down in order to adapt its capacities to the current or expected workload. This is particularly habitual in case of systems deployed on clouds, as cloud-based solutions are often chosen specifically to be easily and frequently scalable.

Both these cases have one thing in common: although preparing and performing them correctly may require careful planning and complete knowledge of all the component relations in the system (especially if we need to keep it running without an interruption), the introduced changes are typically quite restricted and they preserve the general structure of the whole deployment. Updates are usually limited to a single component at a time (unless they introduce some backward incompatibilities between versions) and scaling up and down entails adding or removing instances of components that already exist in the system.

It is also possible sometimes that a major modification, which significantly restructures the system, needs to be performed. In these cases however keeping the system running all the time is rarely required (or possible), so it often happens that such substantial overhauls are performed in three phases: building a new system (separately from the existing one), migrating all the data to it and then tearing the old system down. In fact we should note, that in some environments this kind of procedure is also a common practice in case of much smaller updates, especially if we are not really sure if the updated system will work correctly and we have mastered a sure method of swiftly switching from the old system to the new one.

Deinstallation The last typical reconfiguration of a distributed system is the one which happens at the end of its life, when we simply shut it down and remove all its components. It is certainly the least interesting one. As in this case either way the system is going to be deinstalled completely, there is little reason to care about any constraints being satisfied after it has been stopped. If for some motive however it is required, this situation is symmetric to the initial deployment one.

Reconfigurations in the model

Let us see now how do we encode the reconfigurations in the Aeolus model. First of all we should say that, just as most of the static elements of the model that were described in previous sections are a part of the configuration concept, most of the mentioned dynamic aspects will resurface when defining the reconfigurations. While a configuration is a point in the time of the system's evolution, a *reconfiguration* is the path between two given points, the process of passing from one configuration to another.

We should note, that when we use the expression "a point in time" here in relation with configurations, we do not mean it literally, but in a concrete precise sense related with the Aeolus model. This also applies to our usage of the terms "static" and "dynamic". The system can stay in a certain single configuration for very long periods and it can be very active during that time. Especially if it is completely deployed and running, it will simply do its job, with the software components busy running, carrying out tasks, handling data, communicating with each other, etc. As the Aeolus model's purpose is to model the distributed system at the administrative level, when we mention the flow of time or we refer to static and dynamic aspects, we are reasoning on the system from the administrative perspective, abstracting over most of the details of its functioning in any given stable state. Thus if we do not consider a certain change or action as relevant for the scope which concerns us here (even if it is very important from the point of view of the system's functioning), is will not visible on the model's level as something dynamic. Hence it will rather be a represented in the configuration concept, than as a part of a reconfiguration.s

Having clarified this matter, let us move on to defining the reconfigurations more clearly. As we said, a reconfiguration is the process of transforming the distributed system from one configuration to another. This process can be quite complex, especially if the starting configuration is very different from the final one.

Actions In the Aeolus model we consider each reconfiguration as a sequence of simple steps, basic *actions* that can be performed on a configuration to alter it. The repertoire of available actions, which are in fact building blocks of reconfigurations, is limited to several well defined atomic operations:

- **Adding a component.**

  We add a new component to the configuration. The newly created component is not bound to anything.

- **Deleting a component.**

  We remove an component from the configuration.

- **Binding two ports.**

  We establish a binding between two components, one requiring and one providing, on a certain port.

- **Unbinding two ports.**

  We remove a binding between two components on a certain port.

- **Changing a component's state.**

  We perform a transition in a component's state machine, thus altering its internal state.

- **(*) Changing multiple components' states simultaneously.**

  We perform transitions in state machines of multiple components together (but at most one transition in each component), thus altering simultaneously their internal states.

  Note: This action is optional and discouraged unless necessary, like in the situation depicted in figure 4.10. If it is used in a reconfiguration, we can often slightly modify the way we model the concerned components and replace the combined state change with a sequence of separate state changes (plus maybe some other actions if necessary). However, using a multiple state change may sometimes be a preferable solution if it corresponds better to what is actually happening in the system. A good example of such a situation on the package management level are pre-depend loops which can be found in Debian [29] distributions.

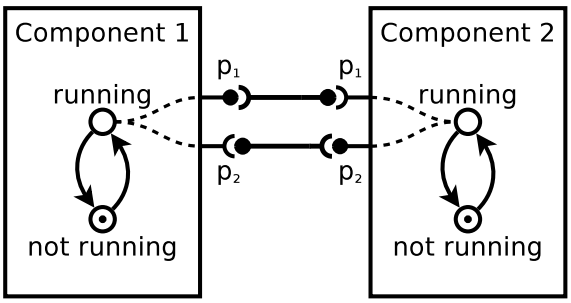


Figure 4.10: The need of the *multiple state change* action: the only way to make both components reach their `running` states without violating the configuration validity is by switching both their states simultaneously.

Feasible reconfigurations

Obviously not all imaginable actions are feasible in every possible configuration. Let us precise more clearly what does it mean.

For example, if a component does not exist in a configuration, we cannot delete it. This kind of thing is clearly an infeasible action as it does not make sense. Intuitively every action that attempts something impossible or breaks the very structure of a configuration (e.g. deletes a bound component and thus leaves some "hanging" bindings behind) should be considered infeasible.

In order to regulate this issue we will specify exactly what all the actions do, so that by definition they cannot break the configuration's structure:

- **Adding a component.**

  This action consists solely of adding a single component to the set of components of the configuration.

  If our system is based on a certain universe, then the new component must be an instance of a component type available in that universe and its initial current state (i.e. the current state it has at the moment of its creation) is determined by the initial state of its corresponding component type.

- **Deleting a component.**

  This action consists solely of removing a single component from the set of components of the configuration.

  The component which is being removed component cannot be involved in any bindings present in the configuration.

- **Unbinding two ports.**

  This action consists solely of adding a single binding to the set of bindings of the configuration.

  If the created binding is on a port `p`, then the its providing component must possess a provide port `p` (which may be active or not) and its requiring component must possess a require port `p` (which may also be active or not).

- **Removing a binding.**

  This action consists solely of removing a single binding from the set of bindings of the configuration.

- **Changing a component's state.**

  This action consists solely of changing the current state of a single component which belongs to the configuration.

  If we are changing a given component's state from state `A` to state `B`, then three conditions must be fulfilled:

  - The component is in the state `A` before the action.
  - Transition from state `A` to state `B` is available in the component's state machine.
  - The component is in the state `B` after the action.

- **(*) Change multiple components' states simultaneously.**

  This action consists solely of changing simultaneously the current states of multiple components which belong to the configuration.

  All the simple state change actions which constitute the multiple state action must be feasible separately according to the rule for changing a single component's state.

Reconfigurations validity

Basing on the feasible actions and valid configurations, we can define a concept of the *reconfiguration validity*. For a reconfiguration to be valid, it must satisfy two conditions:

- Each of its steps must be a feasible action, i.e. an action that makes sense in the context of the configuration at which we have arrived by performing all the previous steps.
- Its initial and final configurations, as well as all the intermediary configurations created by the steps in between, must be valid (as defined by the configuration validity conditions).

Parallelization

We should clarify one detail here. Although we have said, that a reconfiguration is a sequence of actions, this sequence does not always have to be compulsorily executed exactly step-by-step in the reality. Some of the actions can be usually performed in parallel without of violating any system constraints.

It happens often in real distributed systems, that multiple things are done in the same time, especially if it is obvious that they do not interfere with each other. For example setting up ten identical components that do not collaborate with each other (e.g. ten instances of the web application backend) can be easily parallelized, reducing greatly the time required to accomplish this task.

However, if we wanted to represent this kind of parallelization on the model level, we would have to depict the reconfigurations not as simple sequences, but rather as DAGs (Directed Acyclic Graphs) in order to take into account the actual events chronology and the causal dependencies between the actions. This kind of representation is obviously much more complex. Therefore we have decided, that in the scope of the Aeolus model we will always use the simple sequence representation.

This choice does not hinder the possible parallelization of the represented actions in practice (as we can always convert the sequences to DAGs, using our knowledge about the casual dependencies between them, and parallelize the tasks accordingly), we only push this aspect of reconfiguration out of the model.

4.3.10 Summary

In this section we have described thoroughly the core Aeolus model in a non-formal way. First we have given an intuitive definition of each of its parts and specified what are their relationships with each other. In the same time we have outlined the correspondence between the elements and phenomena coming from the real distributed systems and their counterparts in the model. After that we have provided a clearer and more precise description of the Aeolus model's structure as well as its static and dynamic properties. We have also introduced several important concepts like configuration validity, reconfigurations and reconfiguration validity. Now, after grasping the basics of the Aeolus model, we will formalize it completely and take a closer look on some of its interesting proprieties. Finally, we will analyse its expressivity.

4.4 Formalization of the Aeolus model

In this section we will proceed to formally define all the elements of the Aeolus model, which we have described in detail before.

Most of these definitions will match very closely what we have presented in the previous section. Any minor differences are there principally for the reason of convenience (i.e. to make them more comfortable to use in practice). For the same reason throughout this section we will introduce some additional notations, which will help us to refer easily to various parts of the model.

Notations

We will use following basic notations concerning natural numbers:

- $\mathbb{N}^+$ denotes strictly positive natural numbers,
- $\mathbb{N}_\infty$ denotes $\mathbb{N}^+ \cup \{\infty\}$.

Also we will use the symbol $\nrightarrow$ to denote a partial function. For example given two sets E and F, $E \nrightarrow F$ is a partial function from E to F.

Names

In our informal definitions we were assuming that some elements of the Aeolus model have unique names, which let us identify them and refer to them from other parts of the model. In the formal definition we will need to have this kind of global names defined beforehand only for two kinds of entities: ports and components. Thus in the following we assume given the two following disjoint sets:

- $\mathcal{P}$ for ports (or port names),
- $\mathcal{Z}$ for components (or component names).

4.4.1 Universe and configuration

First we will define formally the concepts of the universe and the configuration.

As a universe is basically a set of component types, in order to properly establish its definition we must first define what is a component type. The definition of the universe itself will be then in fact easily merged into the definition of the configuration.

These definitions are mostly based on we have already discussed in detail in section 4.3.7 and section 4.3.8.

Component types

Let us recall, that the component types are basically a sort of blueprints for the actual components. As such, they are an abstract concept and cannot be included directly in a configuration, instead they are instantiated in form of components which are realizing the blueprint. Intuitively we can consider them as Object Oriented Programming classes, while their instances are objects of these classes.

The component type definition contains all the information about the state automaton (including the states, transitions and the initial state) and all the information about the ports (including their respective arities and the activation relation with the states), that every instance of the given component type will have. Formally:

Definition 1 (Component Type)
The set Γ *of* component types *of the Aeolus model, ranged over by* $\mathcal{T}_1, \mathcal{T}_2, \ldots$ *contains 5-ples* $\langle Q, q_0, T, P, D \rangle$*, where:*

- Q *is a finite set of states.*

- $q_0 \in Q$ *is the initial state.*
- $T \subseteq Q \times Q$ *is the set of* transitions *between the states of the state machine.*

  A transition from state A to state B (with $A, B \subseteq Q$*) is available in the state machine if and only if the pair* $\langle A, B \rangle$ *belongs to* T*.*
- $P = \langle \mathbf{P}, \mathbf{R}, \mathbf{C} \rangle$*, with* $\mathbf{P}, \mathbf{R}, \mathbf{C} \subseteq \mathcal{P}$*, is a triple composed of:*
  - $\mathbf{P}$*, the set of* provide *ports;*
  - $\mathbf{R}$*, the set of* require *ports;*
  - $\mathbf{C}$*, the set of* conflict *ports;*

  these are all the ports that the component type possesses.
- *D is a function from Q to 3-ple in* $(\mathbf{P} \nrightarrow \mathbb{N}_\infty) \times (\mathbf{R} \nrightarrow \mathbb{N}^+) \times \wp(\mathbf{C})$*.*

  Given a state $q \in Q$*,* $D(q)$ *returns two partial functions* $(\mathbf{P} \nrightarrow \mathbb{N}^+_\infty)$ *and* $(\mathbf{R} \nrightarrow \mathbb{N}^+)$ *and a subset of* $\mathbf{C}$*, that correspond respectively to:*
  - *the domain of the partial function* $(\mathbf{P} \nrightarrow \mathbb{N}_\infty)$ *indicates the provide ports active in the state q;*
  - *the domain of the partial function* $(\mathbf{R} \nrightarrow \mathbb{N}_\infty)$ *indicates the require ports active in the state q;*
  - *the subset of* $\mathbf{C}$ *indicates the conflict ports active in the state q;*

  The two partial functions associate to the activated ports a numerical constraint indicating:
  - *for provide ports, the port's provide arity (the* maximum *number of bindings the port can participate in);*
  - *for require ports, the port's require arity (the* minimum *number of required bindings to* distinct *components);*

Configuration

We can now define the configurations, that describe systems composed of component instances and bindings that interconnect them.

A configuration, ranged over by $C_1, C_2, \ldots$, is given by a universe containing the component types that are the building blocks of the system, a set of the components deployed in that configuration (each with a type and a current state), and a set of bindings. Formally:

Definition 2 (Configuration)
A configuration *C is a 4-ple* $\langle U, Z, S, B \rangle$*, where:*

- $U \subseteq \Gamma$ *is the finite* universe *of all the component types available in the configuration;*
- $Z \subseteq \mathcal{Z}$ *is the set of the currently deployed* components*;*
- *S is the component* description*, i.e., a function that associates to components in Z a pair* $\langle \mathcal{T}, q \rangle$ *where:*
  - $\mathcal{T} \in U$ *is a component type* $\langle Q, q_0, T, P, D \rangle$
  - $q \in Q$ *is the current component state;*
- $B \subseteq \mathcal{P} \times Z \times Z$ *is the set of* bindings*, namely 3-ples composed by a port, the component that requires that port, and the component that provides it; we assume that the two components are distinct.*

Notations

Let us introduce some useful notations to access various parts of the configuration and the elements of the associated universe.

Component's type and state lookup We write $C[z]$ as a lookup operation that retrieves the pair $\langle \mathcal{T}, q \rangle = S(z)$, where $C = \langle U, Z, S, B \rangle$.

Component type and state pair On such a pair $\langle \mathcal{T}, q \rangle$ we can then use the postfix projection operators `.type` and `.state`:

- $\langle \mathcal{T}, q \rangle$`.type` retrieves $\mathcal{T}$,
- $\langle \mathcal{T}, q \rangle$`.state` retrieves q.

Component type Similarly, given a component type $\mathcal{T} = \langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R}, \mathbf{C} \rangle, D \rangle$, we use projections to (recursively) decompose it:

- $\mathcal{T}$`.states` returns the set of states Q,
- $\mathcal{T}$`.init` returns the initial state q_0,
- $\mathcal{T}$`.trans` returns the set of transitions T;
- $\mathcal{T}$`.prov` returns the set of provide ports $\mathbf{P}$,
- $\mathcal{T}$`.req` returns the set of require ports $\mathbf{R}$,
- $\mathcal{T}$`.confl` returns the set of conflict ports $\mathbf{C}$;
- $\mathcal{T}.\mathbf{P}(q)$ returns the first element of the $D(q)$ triple, indicating the active provide ports and their arities in the state q,
- $\mathcal{T}.\mathbf{R}(q)$ returns the second element of the $D(q)$ triple, indicating the active require ports and their arities in the state q,
- $\mathcal{T}.\mathbf{C}(q)$ returns the third element of the $D(q)$ triple, indicating the active conflict ports in the state q.

Shortcut When there is no ambiguity, we take the liberty to skip one level and apply the component type projections directly to $\langle \mathcal{T}, q \rangle$ pairs. For example, $C[z].\mathbf{R}(q)$ stands for the partial function indicating the active require ports (and their arities) of component z in configuration C when it is in state q.

4.4.2 Configuration validity

We are now ready to formalize the notion of configuration validity (following closely what was already written about this subject in section 4.3.7):

Definition 3 (Configuration validity)
Let us consider the configuration $C = \langle U, Z, S, B \rangle$:

1. *We write $C \models_{req} (z, p, n)$ to indicate that the require port p of component z with associated require arity n is satisfied (i.e. bound to at least n active ports).*

   Formally: there exist n distinct components $z_1, \ldots, z_n \in Z \setminus \{z\}$ such that for every $1 \leq i \leq n$ we have that $\langle p, z, z_i \rangle \in B$, $C[z_i] = \langle \mathcal{T}^i, q^i \rangle$ and p is in the domain of $\mathcal{T}^i.\mathbf{P}(q^i)$.

2. *Similarly for provides, we write* $C \models_{prov} (z, p, n)$ *to indicate that the provide port* p *of component* z *with associated provide arity* n *is not overused (i.e. bound to more than* n *active ports).*

   Formally: there exist no m *distinct components* $z_1, \ldots, z_m \in Z \setminus \{z\}$*, with* $m > n$*, such that for every* $1 \leq i \leq m$ *we have that* $\langle p, z_i, z\rangle \in B$*,* $S(z_i) = \langle \mathcal{T}^i, q^i\rangle$ *and* p *is in the domain of* $\mathcal{T}^i.\mathbf{R}(q^i)$*.*

3. *Likewise for conflicts, we write* $C \models_{confl} (z, p)$ *to indicate that no component is in conflict with the conflict port* p *of component* z *(i.e. all components other than* z *cannot have an active provide port* p*).*

   Formally: for each $z' \in Z \setminus \{z\}$ *such that* $C[z'] = \langle \mathcal{T}', q'\rangle$ *we have that* p *is not in the domain of* $\mathcal{T}'.\mathbf{P}(q')$*.*

All together
The configuration C *is* valid *if for each component* $z \in Z$*, given* $S(z) = \langle \mathcal{T}, q\rangle$ *with* $\mathcal{T} = \langle Q, q_0, T, P, D\rangle$ *and* $D(q) = \langle \mathfrak{prov}, \mathfrak{req}, \mathfrak{confl}\rangle$*, these three conditions hold:*

1.	$(p \mapsto n_p) \in \mathfrak{prov}$	*implies*	$C \models_{prov} (z, p, n_p)$
2.	$(r \mapsto n_r) \in \mathfrak{req}$	*implies*	$C \models_{req} (z, r, n_r)$
3.	$c \in \mathfrak{confl}$	*implies*	$C \models_{confl} (z, c)$

4.4.3 Reconfigurations

We now formalize how configurations evolve from one state to another. These definitions correspond directly to the concepts introduced in section 4.3.9.

Repertoire of actions

First we will define the repertoire of the atomic *actions*, which are the basic steps of the system reconfigurations:

Definition 4 (Available actions)
The set $\mathcal{A}$ *contains the following actions:*

- $stateChange(z, q_1, q_2)$ *where* $z \in \mathcal{Z}$*;*
- $bind(p, z_1, z_2)$ *where* $z_1, z_2 \in \mathcal{Z}$ *and* $p \in \mathcal{P}$*;*
- $unbind(p, z_1, z_2)$ *where* $z_1, z_2 \in \mathcal{Z}$ *and* $p \in \mathcal{P}$*;*
- $new(z : \mathcal{T})$ *where* $z \in \mathcal{Z}$ *and* $\mathcal{T}$ *is a component type;*
- $del(z)$ *where* $z \in \mathcal{Z}$*.*

Note: As we can see, the action of changing multiple components' states simultaneously is not included here. It will be added at the next level, this way we establish it separately as a complex action.

Executing actions

The execution of actions can now be formalized using a labelled transition systems on configurations, which uses actions as labels.

Definition 5 (Actions execution)
Executing actions on configurations is denoted by transitions $C \xrightarrow{\alpha} C'$ *meaning that the execution of* $\alpha \in \mathcal{A}$ *on the configuration* C *produces a new configuration* C'*. The transitions from a configuration* $C = \langle U, Z, S, B\rangle$ *are defined as follows:*

$$C \xrightarrow{stateChange(z,q_1,q_2)} \langle U, Z, S', B\rangle$$
$$\text{and } (q_1, q_2) \in C[z].\texttt{trans}$$
$$\text{and } S'(z') = \begin{cases} \langle C[z].\texttt{type}, q_2\rangle & \text{if } z' = z \\ C[z'] & \text{otherwise} \end{cases}$$

$$C \xrightarrow{bind(p,z_1,z_2)} \langle U, Z, S, B \cup \langle p, z_1, z_2\rangle\rangle$$
$$\text{if } \langle p, z_1, z_2\rangle \notin B$$
$$\text{and } p \in C[z_1].\texttt{req} \cap C[z_2].\texttt{prov}$$

$$C \xrightarrow{unbind(r,z_1,z_2)} \langle U, Z, S, B \setminus \langle p, z_1, z_2\rangle\rangle$$
$$\text{if } \langle p, z_1, z_2\rangle \in B$$

$$C \xrightarrow{new(z:\mathcal{T})} \langle U, Z \cup \{z\}, S', B\rangle$$
$$\text{if } z \notin Z, \mathcal{T} \in U$$
$$\text{and } S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\texttt{init}\rangle & \text{if } z' = z \\ C[z'] & \text{otherwise} \end{cases}$$

$$C \xrightarrow{del(z)} \langle U, Z \setminus \{z\}, S', B'\rangle$$
$$\text{if } S'(z') = \begin{cases} \bot & \text{if } z' = z \\ C[z'] & \text{otherwise} \end{cases}$$
$$\text{and } B' = \{\langle p, z_1, z_2\rangle \in B \mid z \notin \{z_1, z_2\}\}$$

As we can see, in the definition of the transitions there is no requirement on the reached configuration: the validity of these configurations will be considered at the level of reconfiguration plans.

Multiple state change

We want our reconfigurations to be able to contain atomic transitions consisting of multiple simultaneous state changes of different components. To this end, we introduce the notion of *multiple state change*:

Definition 6 (Multiple state change)
A multiple state change $\mathcal{M} = \{stateChange(z^1, q_1^1, q_2^1), \cdots, stateChange(z^l, q_1^l, q_2^l)\}$ *is a set of state change actions on different components (i.e., $z^i \neq z^j$ for every $1 \leq i < j \leq l$). We use $\langle U, Z, S, B\rangle \xrightarrow{\mathcal{M}} \langle U, Z, S', B\rangle$ to denote the effect of the simultaneous execution of the state changes in $\mathcal{M}$: formally,*

$$\langle U, Z, S, B\rangle \xrightarrow{stateChange(z^1,q_1^1,q_2^1)} \dots \xrightarrow{stateChange(z^l,q_1^l,q_2^l)} \langle U, Z, S', B\rangle$$

We should note, that the order of execution of the state change actions does not matter as all the actions are executed on different components.

Reconfiguration / reconfiguration plan

We can now define a *reconfiguration* or a *reconfiguration plan*[5], which is a sequence of actions that transform an initial configuration into a final one without violating validity along the way:

[5]Both terms mean the same thing on the model level. However, as *reconfiguration* suggests more a description of an actual process happening in the system, while a *reconfiguration plan* suggests a plan of such process which can be eventually executed in the system, we rather use *reconfiguration* in the modelling / describing context and *reconfiguration plan* more in planning context.)

Definition 7 (Reconfiguration / reconfiguration plan)
A reconfiguration *or* reconfiguration plan *is a sequence $\alpha_1 \dots \alpha_m$ of actions and multiple state changes such that there exist C_i such that $C = C_0$, $C_{j-1} \xrightarrow{\alpha_j} C_j$ for every $j \in \{1, \dots, m\}$, and the following conditions hold:*

reconfiguration validity *for every $i \in \{0, \dots, m\}$, C_i is valid;*

multi state change minimality *if α_j is a multiple state change then there exists no proper subset $\mathcal{M} \subset \alpha_j$, or state change action $\alpha \in \alpha_j$, and valid configuration C' such that $C_{j-1} \xrightarrow{\mathcal{M}} C'$, or $C_{j-1} \xrightarrow{\alpha} C'$.*

4.5 Properties of the Aeolus model

The principal aim of creating the Aeolus model was to establish a proper frame for our study of the problems related to the automatic deployment of modern distributed systems. A strictly defined and well-formed model was suppose to permit us to apply a formal scientific approach to this domain and hopefuly establish some basic facts about the difficulty of the most commonly encountered challenges.

In particular we wanted to study the computational complexity of the problems associated with deployment automation.

4.5.1 Reconfigurability problem

The main challenge that we will focus on is called the *reconfigurability problem*, which is one of the most fundamental problems related with automated administration of distributed systems. The main question associated with this problem is:

1. *can we get the system from one configuration to another?*

and the supplementary question is:

2. *how can we get the system from one configuration to another?*

First aim: reachability analysis

Let us begin by specifying what the first question really means. We are given a system that is currently in a certain *initial configuration* and there is a certain *final configuration* which we would like to attain. We want to know if it is possible to perform such a *reconfiguration* (i.e. execute a finite sequence of *actions*) on the initial system, that would bring it to the desired final state without violating validity (i.e. executing an action which is not feasible or passing through an invalid configuration) on the way.

Second aim: finding a reconfiguration plan

The second question is simply the constructive version of the first one. We do not only want to know **if** it is possible to get from the first system configuration to the second one, but we also desire to find out **how** exactly can we achieve this objective. Which in fact amounts to finding a reconfiguration plan that corresponds to the desired transformation.

Context: deployment problem

We should now precise the context in which the problem will be posed: what do we exactly mean by the *initial* and *final* configurations.

This problem is supposed to be an approximation of a basic real-life distributed system administration task: deploying a given service in a given distributed system. In this light, the two questions can be refined to a slightly more exact form. The first question is, as before, existential in nature:

1. *can we deploy a given service in the system?*

and the second one is constructive:

2. *how can we deploy a given service in the system?*

In the context of this formulation of the problem, our *initial configuration* corresponds to the given state of "the system": it is thus given and fixed. On the other hand, the *final configuration* is rather associated with the objective: "deploying a given service". Hence it is not exactly pinpointed (i.e. we are not directly given one specific *final configuration* to attain) and it should be in fact considered as a part of the problem.

Inputs and outputs

Now we can define more clearly the inputs and outputs of the reconfigurability problem.

Inputs The inputs are the same for both forms of the problem:

- We are given a system that is in a certain initial configuration. Strictly speaking we are given a configuration $C_{initial}$.
- We are also given a description of the component that needs to be deployed. This description specifies the type of our desired component $\mathcal{T}$ and the internal state q that we want it to be in.

Outputs The outputs differ between the two versions of the problem:

1. In the **existential version** we simply want to know if it is possible to perform a finite sequence of reconfigurations on the initial system, that would give as a system where such a component is present. The answer is thus either **true** or **false**.
2. In the **constructive version** we obtain in certain sense two elements in the answer:
    - we obtain the *reconfiguration plan* that brings us from the initial configuration to a certain valid final one, which contains the desired component,
    - and in the same time we implicitly obtain the *final configuration* itself (which our reconfiguration plan lets us attain).

Request language

As we can see, in the reconfigurability problem we define in a certain fashion the set of acceptable final configurations, and we say that one of them must be attained.

We could see the provided description of the desired component (i.e. its type and state) as some kind of a user request, determining our set of available final configurations. When we specify the component to deploy, it is analogous to fixing this set as: "all the configurations that contain at least one component of that type in that state". It is also equivalent to providing a partial configuration, that any valid final configuration should extend.

Either way we can see here some kind of a request language concept emerging. In this particular case this language remains quite restrained, but we could imagine enhancing its expressivity for other possible applications (like generalization of the reconfigurability problem).

We should also note, that the restriction to *one component in a given state* is not really limiting here. By modifying slightly our universe we can in fact easily encode any given partial final configuration. We simply need to add a dummy provide port enabled only by the required final states (of all the component types that we want to be present in our final configuration) and a dummy component type with requirements on all such provides.

Formalization

We now have all the ingredients to define formally the *reconfigurablility problem*: given an initial configuration (with the corresponding universe of component types), we want to know whether there exists a sequence of reconfigurations leading to a final configuration that includes at least one component of a given component type $\mathcal{T}$ in a given state q.

Definition 8 (Reconfigurability problem)
The reconfigurabilituy problem *has as input:*

- *an initial configuration* $C_{initial} = \langle U, Z, S', B \rangle$,
- *a component type* $\mathcal{T} \in U$,
- *and a target state* $q \in \mathcal{T}.\texttt{states}$.

It returns as output:

- ***true*** *if there exists a reconfiguration plan* $\alpha_1 \dots \alpha_m$ *such that*
  $C_{initial} \xrightarrow{\alpha_1} C_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} C_m$
  and $C_m[z] = \langle \mathcal{T}, q \rangle$, *for some component* z *in* C_m.
- ***false*** *otherwise.*

Achievability problem

The *achievability problem* is a slightly simplified version of the reconfigurability problem. Basically we replace the possibility of providing any initial configuration by always starting from an empty initial configuration: given a universe of component types, we want to know whether it is possible to deploy at least one component of a given component type $\mathcal{T}$ in a given state q.

Definition 9 (Achievability problem)
The achievability problem *has as input:*

- *a universe* U *of component types,*
- *a component type* $\mathcal{T} \in U$,
- *and a target state* $q \in \mathcal{T}.\texttt{states}$.

It returns as output:

- ***true*** *if there exists a reconfiguration plan* $\alpha_1 \dots \alpha_m$ *such that*
  $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} C_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} C_m$
  and $C_m[z] = \langle \mathcal{T}, q \rangle$, *for some component* z *in* C_m.
- ***false*** *otherwise.*

4.5.2 Architecture synthesis problem

There exists also another interesting variation of the reconfigurability and achievability problems, that we call the *architecture synthesis problem*.

The idea behind it is simply to take a reconfigurability or achievability problem instance, disregard completely the questions related to the reconfiguration plan and to focus only on the existence of a suitable final configuration. We do not care in this case if a given final configuration is reachable from the initial one. We only want to know if there exist any valid configuration that satisfies the request (in the existential case) or how does this configuration look like (in the constructive case).

We will write a lot more on the subject of this particular problem and its different versions in the next chapter, therefore we do not include further details here.

4.5.3 Restricting the model

We have studied the reconfigurability and achievability problem in several different contexts, and we have found that the results change significantly if we impose some additional restrictions on our configurations.

Three variants

We have considered three cases, corresponding to different levels of restricting the expressivity of the model (that is directly influencing which configurations do we allow):

- **Aeolus** (or **Aeolus flat**) model is the normal version of our formal model: we can express both functional requirements, non-functional requirements and conflicts.
- **Aeolus core** is an intermediate version of the model, where we can express only functional requirements and conflicts, while non-functional requirements do not exist.

  Specifically it means that we allow only require arities equal one (i.e. no redundancy constraints) and provide arities equal infinity (i.e. no capacity constraints).
- **Aeolus**$^-$ is the most restricted version of the model, where we can express only functional requirements, while conflicts and non-functional requirements do not exist.

  Specifically it means that, in addition to the restrictions imposed on the port arities, we also ban using the conflict ports altogether.

Implementation

We enforce those properties in practice by restricting the possible values associated with the activation of require, provide and conflict ports of all the component types in the universe. Using this mechanism we can limit the allowed require and provide arities and we can switch off the possiblity of activating conflict ports, as required for each model variant.

The exact formal restrictions on the co-domains of particular partial functions of each component type $\mathcal{T}$ are detailed in the table below:

model variant	*co-domain*($\mathcal{T}$.**P**())	*co-domain*($\mathcal{T}$.**R**())	*co-domain*($\mathcal{T}$.**C**())
Aeolus$^-$	$\{\infty\}$	$\{1\}$	$\emptyset$
Aeolus core	$\{\infty\}$	$\{1\}$	$\wp(\mathcal{T}.\texttt{confl})$
Aeolus	$\mathbb{N}_\infty$	$\mathbb{N}$	$\wp(\mathcal{T}.\texttt{confl})$

Alternative implementation

There is also another way of imposing the same set of restrictions on our model, without changing the co-domains. In fact instead of modifying the model directly we could simply modify the configuration validity conditions in each variant of the model.

Doing that we could get the same effects from the other side: we permit all features on the model level (i.e. in every model version we allow all the arities) and simply ignoring checking them from the validity point of view (e.g. every active require port will be satisfied with only one binding, regardless of its actual reqire arity).

This approach is unfortunately slightly more coumbersome in practice, so we stick with the co-domain restriction based one.

4.5.4 Results

We have found some interesting results for all the model variants of the reconfigurability and achievability problems. The discovered properties are summarized in table 4.1.

	full Aeolus model	Aeolus core model	Aeolus$^-$ model
achievability problem	undecidable	decidable, Ackerman-hard	decidable, polynomial
reconfigurability problem	undecidable	decidable, ExpSpace-hard	–

Table 4.1: Aeolus model properties summary.

Property	Source
The **achievability problem** in the full **Aeolus** model is **undecidable**. Therefore the **reconfigurability problem** in the full **Aeolus** model is also **undecidable** (as it is more difficult).	[36]
The **reconfigurability problem** in the **Aeolus core** model is **decidable**. Therefore the **achievability problem** in the **Aeolus core** model is also **decidable** (as it is less difficult).	[33]
The **achievability problem** in the **Aeolus core** model is **Ackerman-hard**.	[34]
The **reconfigurability problem** in the **Aeolus core** model is **ExpSpace-hard**.	[33]
The **achievability problem** in the **Aeolus**$^-$ model is **decidable** and **polynomial**.	[36]

Table 4.2: Aeolus model properties with articles where they were introduced and proved for the first time.

The proofs of these properties can be found in the appropriate articles (see table 4.2) The notations and details may slightly differ, but all the proofs hold. The two first articles describing the Aeolus model and its properties were:

1. *Towards a Formal Component Model for the Cloud* [36]
   Authors: Roberto Di Cosmo, Stefano Zacchiroli, Gianluigi Zavattaro
2. *Component Reconfiguration in the Presence of Conflicts* [33]
   Authors: Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro

There is also a more recent article, which redefines all the previously introduced properties concerning the achievability problem in a more unified way and adds one new property:

3. *Aeolus: a Component Model for the Cloud* [34]
   Authors: Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro

4.6 Expressivity of the Aeolus model

In this section we will present some examples of what we can actually model using the Aeolus approach. First we will discuss in detail our most standard use case: the WordPress farm. Then we will see a few additional minor examples that demonstrate some other intersting aspects of the Aeolus model expressivity.

4.6.1 Distributed WordPress case

The most standard example used throughout this document is the WordPress farm. This is also the use case that we will present and discuss here in detail.

Motivation

This example has many advantages which make it particularly suitable for being a good use case:

- It is a very realistic use case, commonly deployed around the world in many different environments (often in the cloud) and on a various scale.

- On the one hand it is simple enough to be described in a reasonable time and space,
- and on the other hand it is complex enough to showcase most of the important features of the Aeolus model.

Description

WordPress [122] is a popular blogging platform and CMS[6]. It is a web application, written in PHP [99] and executed within a web server such as Apache2 [113] or nginx [91]. In addition, WordPress needs a database connection with a MySQL database system [97] in order to store its data.

Simple WordPress deployments can be realized by installing and configuring all these elements on a single machine. Here we are however interested in bigger and more serious WordPress deployments, capable of supporting higher load and more fault resistant and often deployed in a public (e.g. Amazon EC2) or private (e.g. OpenStack) cloud environment.

This kind of deployments is obviously far more complex than the basic one-machine case. They incorporate multiple WordPress instances installed on different machines (possibly virtual machines). The incoming load is distributed between these instances through some form of a load balancing mechanism:

- One possibility is to balance the load at the DNS [87] level using servers like BIND [64]. They can be configured to answer the DNS requests to resolve a website name with different IPs addresses from a given pool. Under each of these addresses a separate WordPress instance is running.
- Alternatively we can base our load balancing on a HTTP reverse proxy capable of doing that job, used as the website entry point. This solution has and added benefit, because reverse proxy servers such as Varnish [69] can also perform caching, thus contributing significantly to reducing the load passed on to the WordPress instances.

Having multiple WordPress instances installed on different machines does not only help in handling higher load, but also increases the fault tolerance of the whole system: if one WordPress instance crashes or one machine fails, the others will keep on working. However, in such a deployment containing multiple WordPress instances, all of them need to be configured to contact the same MySQL database, to avoid delivering inconsistent results to users.

But, as we have redundancy and load balancing at the front-end side, it is reasonable to have them also at the back-end side (i.e. the database side). One way to achieve that is to use a MySQL cluster and configure the WordPress instances with multiple entry points to it.

Modelling

Let us see how we can approach modelling this use case in the Aeolus model. We will suppose that we are deploying the system on Linux machines (running for example Debian) in a private cloud environment like OpenStack.

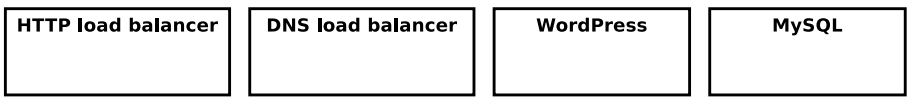


Figure 4.11: The four component types of the WordPress farm use case.

Software components First we will discuss the division of the whole distributed system into components. We will focus on modelling the serious case of the WordPress scenario, thus we will want to have a full system with a load balancer in the front, multiple WordPress instances in the middle and a MySQL cluster behind, also with multiple instances.

We choose to have four different component types available:

[6]Content Management System

- DNS load balancer: a load balancing component based on a DNS server such as BIND.
- HTTP load balancer: a load balancing component based on a reverse HTTP proxy server such as Varnish.
- WordPress: an instance of the WordPress blogging platform, together with an Apache2 web server and the Apache2 PHP module.

  We have decided to put all those elements in the same component, because only when they are all installed on the same machine and configured to work together they will really deliver what we would call the "WordPress service".
- MySQL: an instance of the MySQL database, forming a MySQL cluster with all the other MySQL instances in our system.

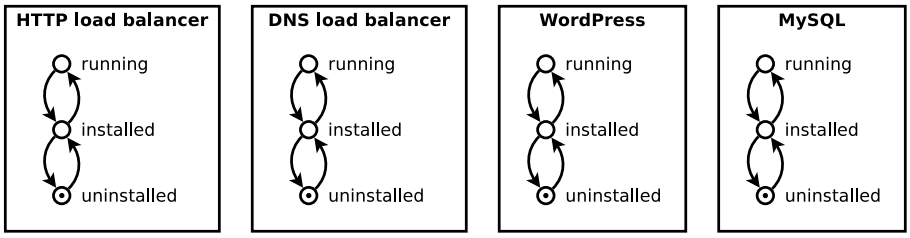


Figure 4.12: Our component types fitted with state machines in order to model their internal state and its possible evolution. Dots mark the initial state of each automaton.

State machines Second, we should fit all our component types with state machines which represent their internal state and its possible evolution. In this case the most common state machine with three standard states, `uninstalled`, `installed` and `running`, and the usual transitions between them, seems appropriate enough.

In each of the state machines the transitions mean more or less the same thing in term of the related administrative actions:

- **from** `uninstalled` **to** `installed` :

  Install the required software packages using the package manager
  (e.g. `sudo apt-get install mysql-server`).
- **from** `installed` **to** `running` :

  Run the appropriate service.
  (e.g. `/etc/init.d/mysql start`)
- **from** `running` **to** `installed` :

  Stop the appropriate service.
  (e.g. `/etc/init.d/mysql stop`)
- **from** `uninstalled` **to** `installed` :

  Remove the software packages installed before.
  (e.g. `sudo apt-get remove mysql-server`).

Ports Now the time has come to add ports to our component types. Let us consider the question of the relations between the components in our distributed system that these ports should model: the functional requirements, non-functional requirements and conflicts.

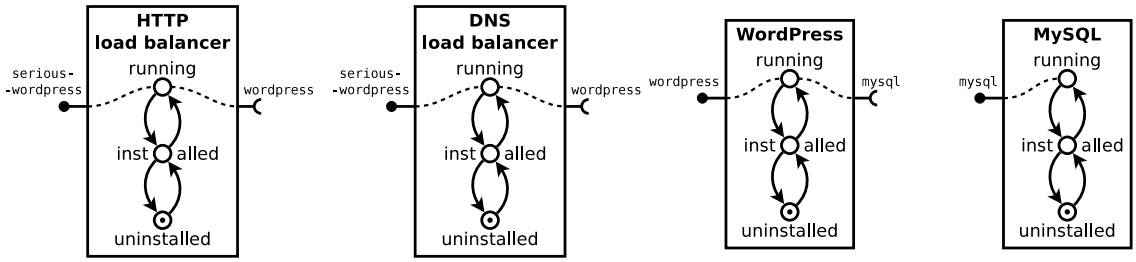


Figure 4.13: Our component types refined through modelling the functional requirements present in the system as require and provide ports.

Functional requirements First we determine the functional requirements and model them using require and provide ports:

- Each WordPress instance needs a connection with a MySQL Cluster entry point, i.e. a MySQL instance. This connection is only required when the WordPress is in the `running` state. Moreover the MySQL component on the other side needs to be also `running` in order for this connection to work.

  Thus the WordPress component type will have a require port `mysql`, active only in its `running` state, and the MySQL component type will have the complementary provide `mysql` port, also active only in its `running` state.

- Both load balancers (the DNS-based one and the HTTP-based one) need to know about at least one WordPress instance (up and `running`!) to load balance it. Of course they need this only in order to be `running`.

  Thus both our load balancer component types will have a require port `wordpress`, active only in their `running` state, and the WordPress component type will have the complementary provide `wordpress` port, also active only in its `running` state.

  Note: This may be slightly counter-intuitive. Without a specific context we would tend consider a load balancer as a very universal kind of software component, which should not have precise requirements like that. It can load balance any web application, so why would it always require a WordPress instance?

  Well, in the context of our WordPress farm distributed system the purpose of a load balancer is indeed to balance the load between many WordPress instances. And we want to model exactly this kind of relation here, seen from the system administration point of view in the context of the whole distributed system.

  If there were multiple different component types to load balance with the same load balancer, then we would probably have to change the way we model the load balancer components (i.e. their state machines and ports) or add multiple variants of load balancers (i.e. one for each service), depending on what kind of effect we want to achieve.

- Finally, we need to model the fact that our load balancers are in fact providing a "serious, load-balanced and robust WordPress service" to the external world. To this end they will both be equipped with a provide port `serious-wordpress`, active only in their `running` states.

Non-functional requirements Now we can elaborate these by adding some port arities in order to express non-functional requirements present in our distributed system:

- We would like each WordPress instance to be connected to more than one MySQL server in the cluster at any given time. This way in case if one MySQL server fails, the WordPress will still have the other one immediately available.

  We will model this non-functional requirement by giving a require arity of 2 to the require `mysql` port of our WordPress component type.

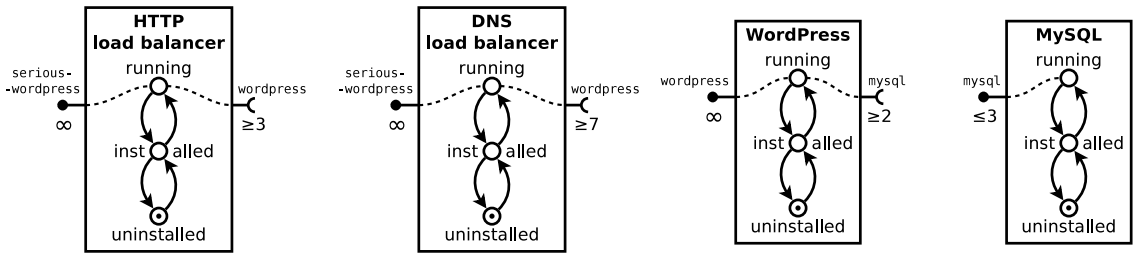


Figure 4.14: Our component types refined through modelling the non-functional requirements present in the system as require and provide arities.

- On the other hand we do not want any of our MySQL servers to be overloaded by too many WordPress instances connected to them and sending requests at the same time. We would like to set an upper bound here an allow a maximum of three WordPress instances simultaneously connected to each MySQL instance.

  We will model this requirement by setting the provide arity of the MySQL component's `mysql` port to 3.

  Note: These two constraint incidentally fix the ratio between the WordPress and MySQL components in our system and thus define how it scales up or down.

- Now we should define the require arities on the `wordpress` port for the two load balancers. This is a slightly different kind of design decision, as it basically corresponds directly to the number of WordPress instances which we will have in our system. Therefore it is a kind of an arbitrary *deployment policy* decision, which depends entirely of what kind of load and fault tolerance level do we aim for.

  We decide that if we use the DNS-based load balancer, simply distributing the load evenly between all the IP addresses in its pool, we will require seven WordPress instances to handle all the load. However, if we use the load balancer based on the reverse HTTP proxy, we only require three WordPress instances, because of the estimated gain from the caching the load balancer performs.

  We will model this by giving a require arity of 7 to the require `wordpress` port of the DNS load balancer component type, and a require arity of 3 to the same port of the HTTP load balancer component type.

- When we consider what should we do with the other side of this relation (the `wordpress` port provided by the WordPress component type), we decide that there is no reason to limit the number of load balancers that can load balance a single WordPress instance. We only aim for one load balancer here, and if there are more, they will probably only serve as backup.

  Thus we give set the provide arity of the WordPress component's `wordpress` port to infinity.

- Finally, we should decide which arity to give to our both load balancers' provide `serious-wordpress` ports. Thanks to the appropriate difference in the number of the WordPress backends behind them, they should both provide a `serious-wordpress` service on comparable level, so their provide arities on this port should probably be the same. Other than that, we have no reason[7] to choose any particular value, as there are no component types requiring this port, only a vague "external world".

  Therefore we fix the provide arity of the `serious-wordpress` ports of our both load balancers to infinity.

[7] Unless we consider will consider this model in the context of the reconfigurability or achievability problem.

Conflicts After taking care of modelling the functional and non-functional requirements, the time has come to see if we have some conflicts present in our distributed system:

- There is a self-conflict concerning one of our component types indeed. As load balancer based on DNS is functioning in fact as a DNS server, and usually we do not want to have more that one primary DNS server in our network, we will model it a a singleton component: simultaneously there can be at most one of them running in our system.

  To this end, we fit it the DNS load balancer component type with two additional ports, both active only in its `running` state: one conflict port named `dns` and one provide port (with an infinite provide arity) also called `dns`.

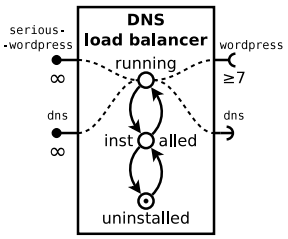
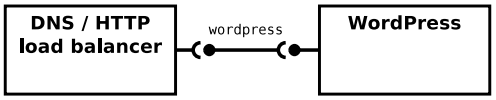
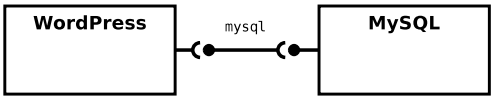


Figure 4.15: Our only component type with a conflict.



(a) The binding between a load balancer instance (HTTP or DNS based) and a WordPress instance on port `wordpress`.



(b) The binding between a WordPress instance and a MySQL instance on port `mysql`.

Figure 4.16: The collaboration between instances of our component types, represented in the model in form of bindings.

Bindings Now we should probably give some more exact meaning to the bindings in our system and various phases of their life: creation, existence and removal.

WordPress and MySQL The binding between a WordPress component and a MySQL component on the `mysql` port:

1. **Creation**: The WordPress instance is configured to connect to a given machine (IP address) and use a database with a certain name; details like user name and password are prepared and synchronized between the two in order to let the communication happen (e.g. the system administrator generates a random user name and password and sets it up on both sides).

2. **Existence**: The WordPress instance is sending queries to the MySQL instance and the MySQL instance is responding.

3. **Removal**: Both instances are reconfigured in order to forget about each other (i.e. basically what was done during the creation stage is reversed).

DNS / HTTP load balancer and WordPress The binding between the DNS or HTTP load balancer component and a WordPress component on the `wordpress` port:

1. **Creation**: The load balancer is reconfigured to add this WordPress instance (i.e. probably simply the IP address of the instance's machine) to the instances that it is currently load balancing. Nothing happens on the WordPress instance side (as it is configured exactly in the same way if load balanced or not).

2. **Existence**: The load balancing takes place: basically the load balancer is distributing the incoming traffic among the available WordPress instances in some way (and the load balancer based on the reverse HTTP proxy is also performing caching).

3. **Removal**: The load balancer is reconfigured to remove this WordPress instance from the instances that it is currently load balancing.

The full example

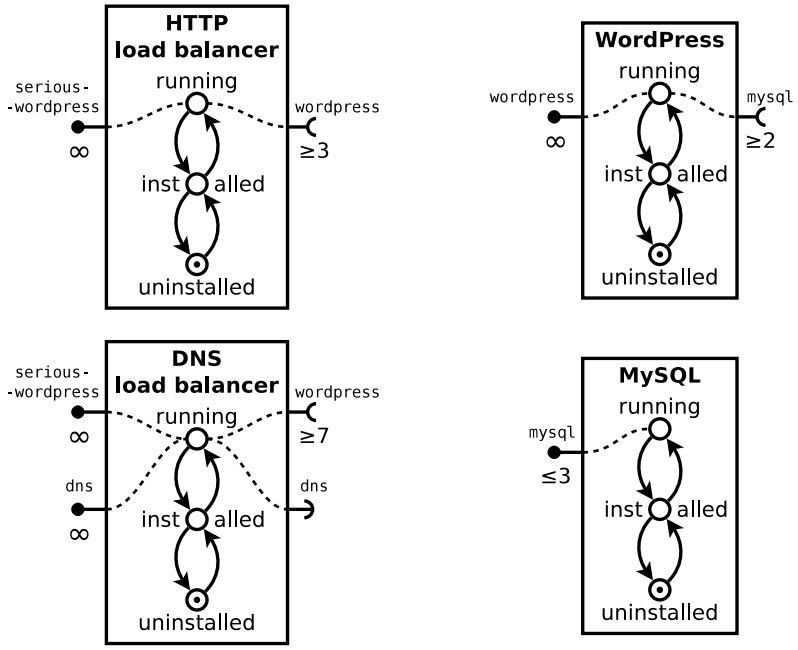


Figure 4.17: Our complete universe of component types.

Let us consider a use case of deploying a WordPress farm instance as modelled here, as in the achievability problem. The initial configuration of our system is empty and the final one that we want to attain contains at least one component providing the `serious-wordpress` port.

As this is a quite simple case, it is very easy to design such a final configuration by hand. It contains one HTTP load balancer, three WordPress components and two MySQL components; all of them in their `running states`. There are three bindings on the `wordpress` ports between the single HTTP load balancer instance and each the three WordPress instances; and there is a total of six bindings between the WordPress instances and the MySQL instances: two bindings for each WordPress. All this is illustrated on the figure 4.18.

We can see that, if we respect the Aeolus model configuration and reconfiguration validity conditions, all the actions which we can make and all the possible states that we can attain are correct. There are many different sequences of actions that can take us from the initial configuration to the final one, but we can be sure that all the reconfigurations valid in the Aeolus model would correspond to safe and correct actions and states on the real distributed system level. We detail here a few examples of the guarantees that are ensured by the model:

- We will never see the load balancer running until enough load balanced WordPress instances are running and thus it is capable of supporting the expected "serious" load.
- A MySQL instance can never be stopped (e.g. for maintenance) if at least one WordPress instance is potentially using it.
- Also a WordPress instance will never be started before it has been configured to connect with two different MySQL instances. Therefore, when it is running, it will always have at least two alternative database connections available in case of problems.

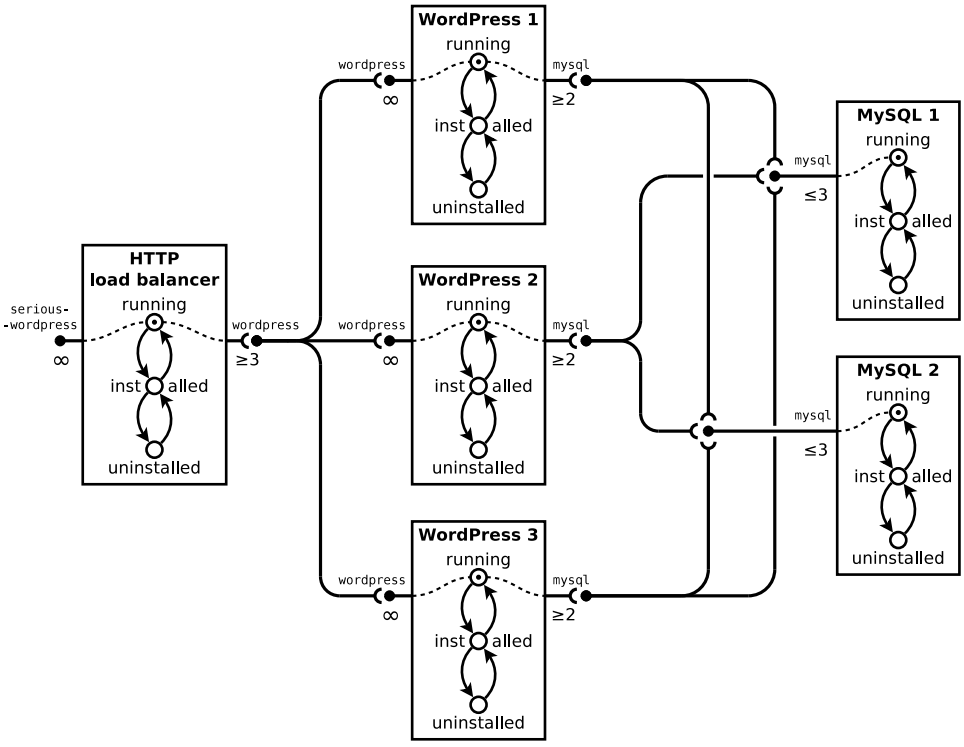


Figure 4.18: Fullly deployed WordPress farm use case.

CHAPTER 5

SYNTHESIS OF A COMPONENT ARCHITECTURE

5.1 Motivation

In the previous chapter, we have presented a formal model for modern distributed systems that is expressive enough to cover most common use cases, and yet simple enough to allow a formal analysis of its main properties.

We have seen that in the full Aeolus model the key problem of reconfiguration is undecidable, and becomes decidable, but with very high complexity, for some simplified versions of the model.

The Aeolus model is actually a simplified abstraction of the complexity present in real systems, hence these limiting results tell us that, despite the hype which is commonplace in the Cloud world, full automation is a goal which is impossible to achieve in general; this explains why current tools and technologies available in the real world are so limited.

Still, there is no reason to stop our inquiries after reaching this conclusion. There are many other approaches that we can try in order to progress and reach some useful results. In this chapter we will isolate a single aspect of the problem and try to find a way to solve it more efficiently than what we can do for the whole problem.

My contribution

The work presented in this chapter has not been officially published (yet). I participated in the conceptual part of the work, together with the Aeolus project team in Paris: Michael Lienhardt, Roberto Di Cosmo, Ralf Treinen and Stefano Zacchiroli. Moreover, I was the main author of a prototype tool that realized these theoretical ideas in practice (which later evolved into the Zephyrus tool).

5.2 The architecture synthesis problem

The challenge that we are going to focus on in this chapter is the *architecture synthesis problem*. In order to define this problem in a more straightforward manner we will introduce a special adapted version of the Aeolus model, which suits this particular problem better.

5.2.1 Problem definition

Differently than in the reconfigurability or achievability problems, in the architecture synthesis problem we concentrate solely on finding a final system configuration which is valid and satisfies a given request. We do not attempt to find a reconfiguration plan which could let us bring the system from its initial configuration to the final one. In scope of the architecture synthesis problem we do not even care if such a plan exists, thus if the found final configuration is reachable in any way from the initial one.

Inputs

The inputs of the architecture synthesis problem are:

- An *initial configuration*,
- A *universe*,
- A *request* defining the properties that we demand of the desired final configuration.

In many cases we presume that the initial configuration is valid with respect to the input universe, but this is not obligatory.

Output

The output of the architecture synthesis problem is a *final configuration* (sometimes also referred to as the *desired configuration*), which:

- must be valid with respect to the input universe
- and must satisfy the input request.

5.2.2 Discussion

When considering the architecture synthesis problem on its own, one might doubt if trying to solve it would get us anywhere. Will in practice finding a desired final configuration help us in any way, when we do not know how to reach it, nor even if it can be reached at all?

Orthogonality with reachability analysis

Obviously, if we manage to find a final configuration reachable from the initial one, then we could proceed to searching for a right reconfiguration plan afterwards. In other words, this way we would be basically dividing the whole problem into two easier orthogonal sub-problems, which could be solved independently.

Unfortunately this is not always the case, as in this approach we consciously give up trying to find a final configuration which is surely reachable. As a result, we have to consider two possibilities:

- if we happen to end up with a final configuration that is actually reachable, the search for a suitable reconfiguration plan can be performed and succeed, thus solving the whole problem;
- if we find one that is not reachable, the search for the reconfiguration plan will fail, but this does not mean that there are no other correct final configurations which are reachable[1].

Therefore these two sub-problems (i.e. architecture synthesis and reachability analysis) remain still inter-dependent in this context.

[1] In order to have this certainty we would have to iterate the process over and over again until either we find a correct and reachable final configuration or we exhaust all possibilities (when iterating we must assure that each time we exclude all the already considered final configurations).

Utility

However, we believe that this approach is a fairly useful idea.

Valuable standalone First, taking into account all the complexity of inter-component relationships which can exist in a modern distributed system, designing a correct system configuration that fulfils given constraints is often already a difficult and tedious task even without considering all the issues related to deploying it. So there is certainly some utility in studying this particular sub-problem separately and trying to solve it automatically, as the capacity to find a requested configuration, even without getting the associated reconfiguration plan, is already quite valuable.

Outside-the-model reconfiguration Moreover, we may simply assume that when the desired system configuration is already found, somebody will always be able to figure out how to realize it. Even if this configuration happens to not be theoretically reachable (e.g. when it contains a causal dependency cycle between two components), there may exist some "dirty" methods, far too low-level to encode within our formal model, which permit us to deploy it anyhow. We may imagine that we have at our disposal a stubborn system administrator with a screwdriver, who is capable of disassembling our system to little pieces and reassembling it directly into the new shape, if all the conventional methods fail.

Restricting input Yet another idea is to decide directly that we restrict the inputs of the architecture synthesis problem that we consider to such initial configurations, universes and requests which can give us only reachable final configurations. In practice this is not a completely implausible assumption, as it seems reasonable to expect that most of the well designed distributed systems which exist in the real world should naturally have this property (i.e. it should be possible to find a way to reconfigure such a system from one correct state to another).

Conclusion

Either way, for better or worse, in this approach we simply assume that if we can find a correct final configuration, then there must be some way to reach it. Hence for now we completely abandon the question of reachability and should not discuss it any more in this chapter.

5.2.3 Stateless model

Now we will define formally the Aeolus stateless model, which was designed specifically to describe instances of the architecture synthesis problem.

Introduction

When we decided to focus on solving the architecture synthesis problem on its own, we noticed that the full Aeolus model described in the previous chapter is unnecessarily rich for this purpose. As in the context of this particular problem we are interested only in two configurations of the system, the initial and final one, and as the question of reachability is out of our scope for now, we do not need to care any more about what happens between these two configurations. We can thus simplify the model greatly by taking the time flow out of the equation and hence removing all the temporal and causal features.

Unnecessary parts As now we operate only on individual system states and the problems related with the state changes are not relevant to us, we can strip our model of everything that is related to the system reconfigurations. This not only means that we drop the whole part of the Aeolus model concerning the actions and reconfiguration plans, but also that we can get rid of the elements that rely on it, namely the transitions in the components' state machines.

The whole point of fitting components with state machines was to model how their properties and behaviour vary when the internal state shifts. In the current context it becomes aimless, because we focus only on fixed points in time: in each single moment the component remains in a certain specific state, so

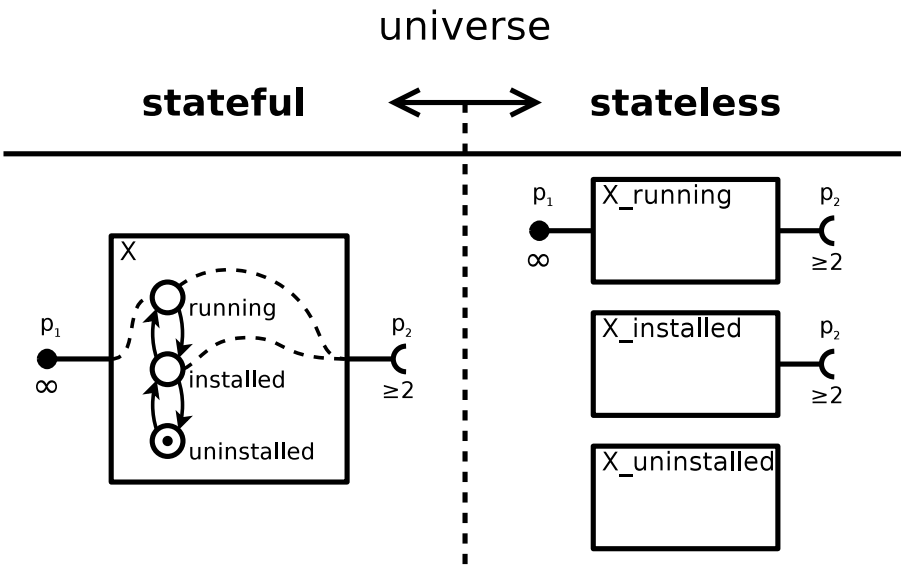


Figure 5.1: Converting an example stateful component `X` to three corresponding stateless components: `X_uninstalled`, `X_installed` and `X_running`.

the information related to all the other states is useless. From this point of view the different "versions" of a single component which evolves over time (corresponding to its different states) are not really associated with each other any more, hence they can be as well considered as completely different, not affiliated components.

Correspondence Because of all that, we will define a new stateless version of the Aeolus model where each stateful component type in a certain state becomes a separate stateless component type. Thus each component type from the stateful model corresponds to a whole set of component types in the stateless model (each stateless component type corresponding to exactly one of the states in the stateful component type's state machine). We can see an example of such an equivalence in figure 5.1.

Formalization

Let us now finally formalize the stateless Aeolus model. The following formal definitions will be used for other purposes than the ones introducing the stateful Aeolus model, so they are constructed in a slightly different fashion. Nevertheless, as main principles behind both formal models are similar, many of these new definitions will be quite close to the ones presented in chapter 4. In order to avoid confusion we will redefine everything here (including the notations) from the scratch.

Notations

- We will use following basic notations concerning **natural numbers**:
  - $\mathbb{N}$ denotes natural numbers with zero,
  - $\mathbb{N}^+$ denotes strictly positive natural numbers,
  - $\mathbb{N}^+_\infty$ denotes $\mathbb{N}^+ \cup \{\infty\}$.
- We will define more strictly the partial functions as objects called **mappings**:

  Given two sets X and Y, we call a *mapping* $f : X \mapsto Y$ any function f such that its domain $dom(f)$ is a finite subset of X and whose image $f[X]$ is included in Y (i.e. $dom(f) \subseteq X \wedge f[X] \subseteq Y$).

- In order to work on various elements of our model more comfortably in practice, we will also define a **tuple lookup** operation (which is in fact a meta-notation):

  Given a tuple $T = \langle \ell_1, \dots \ell_i \rangle$, we note $T.\ell_i$ the lookup operation that retrieves the element ℓ_i from the tuple T.

Names

In the stateless variant of the Aeolus model the connection between the universe and the configuration will look somewhat different that in the stateful one. Now we need to give unique names not only to the ports and components, but also to the component types, so that we can reference all of them. Therefore in the following we suppose given three infinite disjoint sets:

- a set of port names $\mathcal{P}$, ranged over by p_1, p_2, etc;
- a set of component type names $\mathcal{T}$, ranged over by t_1, t_2, etc;
- a set of component names C, ranged over by c_1, c_2, etc.

Universe

Now we can define the structure of a *stateless component type* and a *stateless universe*. As the component types have been stripped of their state automata and the whole port activation mechanism is no longer needed, their definition becomes much less complex than it was in the stateful version of the Aeolus model:

Definition 10 (Stateless Component Type)
A component type *is a triple* $\langle \mathbf{P}, \mathbf{R}, \mathbf{C} \rangle$*, where:*

- $\mathbf{P} : \mathcal{P} \mapsto \mathbb{N}^+_\infty$ *is a mapping defining the ports provided by the component type (the domain of the mapping) with their provide arity (the values for each port name);*
- $\mathbf{R} : \mathcal{P} \mapsto \mathbb{N}^+$ *is a mapping defining the ports required by the component type (the domain of the mapping) with their require arity (the values for each port name);*
- $\mathbf{C} \subset \mathcal{P}$ *is the finite set of conflict ports of the component type.*

We note Γ *the set of all component types.*

Definition of the *stateless universe* follows directly (and now it is mentioning the names of component types which it contains):

Definition 11 (Stateless Universe)
A universe $\mathcal{U} : \mathcal{T} \mapsto \Gamma$ *is a finite mapping defining the set of available component types (the image of the mapping) with their names (the domain of the mapping).*

Notations

- Given a universe $\mathcal{U}$, we note:
  - $\mathcal{U}_{dt}$ the set of names of the component types available in the universe $\mathcal{U}$:

    $$\mathcal{U}_{dt} \triangleq dom(\mathcal{U})$$

  - $\mathcal{U}_{dp}$ the set of names of ports used in the universe $\mathcal{U}$:

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

- Moreover:
  - $\mathcal{U}\mathtt{R} : \mathcal{P} \to \wp(\mathcal{U}_{dt})$ gives us all the *requirers* of the given port, i.e. the set of all the component types that belong to the universe $\mathcal{U}$ and require the port in parameter:

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

  - $\mathcal{U}\mathtt{P} : \mathcal{P} \to \wp(\mathcal{U}_{dt})$ gives us all the *providers* of the given port, i.e. the set of all the component types that belong to the universe $\mathcal{U}$ and provide the port in parameter:

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

  - and $\mathcal{U}\mathtt{C} : \mathcal{P} \to \wp(\mathcal{U}_{dt})$ gives us the *conflicters* of the given port, i.e. the set of all types component types that belong to the universe $\mathcal{U}$ and conflict with the port in parameter (i.e. possess the corresponding conflict port):

$$\mathcal{U}\mathtt{C}(p) \triangleq \{t \mid t \in \mathcal{U}_{dt} \wedge p \in \mathcal{U}(t).\mathbf{C}\}$$

Configuration

Now the time has come to introduce *components* and *configurations*. In the stateless model a *component* is just an instance of a component type, with no internal structure at all. A *configuration* C is a set of components with their bindings:

Definition 12 (Stateless Configuration)
A configuration C *is a pair* $\langle W, B \rangle$*, where:*

- $W : C \mapsto \mathcal{T}$ *is a mapping form component names* C *to component type names* $\mathcal{T}$*, defining the set of names of components which are present in the configuration (the mapping's domain) and stating for each component its type (the mapping's values);*
- $B \subset \mathcal{P} \times dom(W) \times dom(W)$ *is the set of* bindings *existing in the configuration, which are triples consisting of, respectively:*
  - *a port name,*
  - *the name of the component that requires that port*
  - *and the name of the component that provides it.*

We note Θ *the set of all configurations.*

Notations

- Given a configuration $C = \langle W, B \rangle$, we note:
  - C_c the set of names of all the components in that configuration;

$$C_c \triangleq dom(W)$$

  - and C_t the set of names of all the types of components in that configuration.

$$C_t \triangleq \{t \mid \exists c \in C_c,\ W(c) = t\}$$

- Given a component name $c \in C_c$, we note $C(c)$ the name of the type of the component c in the configuration C.

$$C(c) = W(c)$$

- Given a component type name $t \in C_t$, we note $C(t)$ the set of names of all the components in C whose type is t in the configuration C.

$$C(t) = \{c \mid C(c) = t\}$$

Remark about stateless configuration properties

We should notice that, differently to the stateful configuration, in our definition of the stateless configuration both components and component types are referenced indirectly: not by the objects themselves, but by their names. Because of that, as the mappings associating components to component type names and component type names to component types are now separated (i.e. there is no direct mapping from components to component types), the universe is not directly included in the configuration any more.

Thanks to fact that the universe is not integrated into a configuration, we gain a little bit more flexibility in combining configurations with universes. On the other hand because of this structural change in the definitions, some part of what was formerly considered as inherent properties of configurations will now need to be shifted to the definitions concerning the configuration validation (with respect to a given universe) and expressed explicitly.

Nonetheless, we can see that the bindings which belong to a configuration remain defined as a set, so the binding unicity principle (the rule, that there can be at most one binding from a given provided port to a given required port between two given components) is still a structural property of the model.

5.2.4 Request language

Having defined the stateless version of the Aeolus model, which is much more adapted for describing the architecture synthesis problem than the stateful version used throughout the previous chapter 4, we can move on to specifying the request language.

Input

As we have already established, the inputs of the architecture synthesis problem consist of three elements: an initial configuration, a universe and a request defining what do we want from the desired final configuration. We will also call that request a *specification*, because it serves us to specify additional constraints which the final configuration has to satisfy: the solution of the architecture synthesis problem, our desired configuration, must not only be valid, but also fulfil the provided specification.

The specification will be expressed using a strict request language, which will permit us to formulate in a clear and precise manner the constraints that we want to impose. The request language that we will introduce here is quite simple and based on the previous idea of a "partial configuration to extend", where the final configuration should contain at least everything that is present in the given partial one. Although we could easily imagine using a much more elaborate language, allowing us to specify many different complex properties, for now we will restrain ourself to this simple one, which is already sufficient for posing many interesting problems.

Expressivity

In the deployment problem we were concentrating on a single instance of a certain component type (in a specific state), that was supposed to be present in the final configuration. Here we will extend this approach a little: instead of asking for one component of a given type, we will ask for at least a given number of components or ports of given types. Our request language will therefore let us specify that:

- At least `n` **components of a given type** should be present in the final configuration.
- Total provide arity of at least `n` of a certain **port** should be provided by the components throughout the final configuration.
- Any conjunction of these two conditions (for different component types or ports).

Port arity

As we can see, these constraints are quite simple and indeed correspond well to the idea of providing a partial configuration which should be extended. There is only one slight doubt that can arise when we consider them.

While the "at least n components present" condition makes sense intuitively and is a reasonable request that one may want the system to fulfil, the "at least n port arity provided" is less obvious. Its primary purpose is to allow us, instead of asking directly for a certain component type, to rather ask for *any* component that provides the service represented by a particular port.

This allows us to gain more flexibility in our requests, and at the same time keeping the request language as simple as possible. If we wanted to introduce a constraint that gives us a similar expressivity by enumerating directly a set of component types to choose from (e.g. at least one component of type A or one component of type B), then we would need to include disjunctions in our request language, thus making it more complex. Moreover it would not be equivalent to providing a partial configuration any more.

However, we should notice that, this condition, as it is formulated, does not say anything about the presence or absence of bindings connected with these ports. Hence the provide ports that we ask for do not have to be necessarily free, they can all be bound. Therefore, although this obviously guarantees us that a certain number of components providing these ports (summarily at least in the demanded arity) are present in the configuration, it must be used carefully, because the ports themselves may be already used by other components.

Formalization

Now, in order to have everything completely clear and well defined, let us formalize the abstract syntax of the request language that we will use to express specifications:

S	::=	**true**	$\mid$ $S \wedge S$	$\mid$ $e \geq n$		Specification
e	::=	$N(t)$	$\mid$ $N(p)$			Expression

This syntax lets us express the constraints imposed on a given configuration by the specification in form of a conjunction of comparisons of integer values (constants and variables). The used notations represent:

- S is the whole specification. It can be either:
  - **true** so it does not introduce any constraints;
  - a conjunction of two specifications ($S \wedge S$), which both have to be fulfilled in order for the conjunction to be true;
  - a comparison of a *specification expression* and a natural number ($e \geq n$), which represents a constraint that must hold in the configuration.
- e is a *specification expression*, which can be attributed a numerical value in the context of a particular configuration:
  - the variable $N(t)$ stands for the number of components of the given type t in the synthesized configuration;
  - the variable $N(p)$ stands for the total arity of ports of the given name p provided together by all the components present in the configuration.

We will define formally the meaning of a specification when we discuss the configurations validity with respect to a specification.

5.2.5 Stateless configuration validity

We have discussed one by one all the parts of the architecture synthesis problem and now the time has come to start bringing them together. As we have established the stateless version of the Aeolus model and introduced the request language for our specifications, we possess all the pieces required to specify the inputs of the problem. Let us now focus on defining precisely, what are the properties of the desired final configuration that we are looking for. For that purpose we will use the concept of the configuration validity, refined in order to work in this new context.

Informal definition

When given as input of the architecture synthesis problem:

- an initial configuration,
- a universe
- and a specification,

then a certain final configuration is a valid solution of that problem if it both:

- is valid with respect to the given **universe**:
  - it contains only components which are instances of the component types of the universe.
  - its components and bindings fulfil all the necessary functional and non-functional requirements.
- fulfils the given **specification**.

Validity with respect to universe

In the context of the architecture synthesis problem, the final configuration that we are looking for is always supposed to be a stable, fully operative system, therefore we will only accept solutions that satisfy both the functional and non-functional requirements defined by the given universe.

In difference with the stateful Aeolus model configuration validity, here we know that all the ports (provide, require and conflict) are always currently active, because the non-active ports are simply not represented in the stateless model. When we take into account both this fact and the binding unicity principle (which is a structural property of the stateless configurations, as bindings are represented as a set), we may simplify the validation conditions greatly: in comparison to the stateful model, is is enough to count the incoming and outgoing bindings in order to verify if the non-functional requirements hold (i.e. if the provide and require arities are respected), as we do not have to check if the port on the other side of the binding is actually active.

Definition 13 (Configuration validity w.r.t. universe)
Suppose given a configuration $C = \langle W, B \rangle$ and a universe $\mathcal{U}$.

C is valid *w.r.t. $\mathcal{U}$ if for all the components $c \in C_c$ present in the configuration:*

- *the component's type $t = C(c)$ belongs to the universe $\mathcal{U}$;*

$$C(c) \in \mathcal{U}_{dt}$$

- *there are no bindings on port p in which the component is the providing side if it does not provide the port p;*

$$\forall p \in \mathcal{P} \setminus dom(\mathcal{U}(t).\mathbf{P}),\ \{c_{req} \mid (p, c_{req}, c) \in B\} = \emptyset$$

- *the component's provide ports are not overused;*

  (i.e. the number of components that the component is bound to on port p as a providing component must not be bigger than its provide arity on port p)

$$\forall p \in dom(\mathcal{U}(t).\mathbf{P}),\ \left|\{c_{req} \mid (p, c_{req}, c) \in B\}\right| \leq \mathcal{U}(t).\mathbf{P}(p)$$

- *the component's require ports are satisfied;*

  (i.e. the number of components that the component is bound to on port p as a requiring component must be at least as big as its require arity on port p)

$$\forall p \in dom(\mathcal{U}(t).\mathbf{R}),\ \left|\{c_{prov} \mid (p, c, c_{prov}) \in B\}\right| \geq \mathcal{U}(t).\mathbf{R}(p)$$

- *and there can be no conflicts.*

  (i.e. if a component has a conflict port p then it is the only component in the configuration that can also provide the port p)

$$\forall p \in \mathcal{U}(t).\mathbf{C},\ \left\{c_{confl} \mid c_{confl} \in C_c \wedge C(c_{confl}) \in \mathcal{U}\mathrm{P}(p)\right\} \subseteq \{c\}$$

Validity with respect to specification

Another essential property of the desired final configuration is that it should satisfy the given request. The request, defined using the *abstract specification syntax*, specifies additional constraints imposed on the final configuration. It may be interpreted as a partial configuration that any valid final configuration has to contain and extend.

Definition 14 (Configuration satisfying the specification)
Given a universe $\mathcal{U}$, the configuration C satisfies a specification S if and only if $C \vdash S$ can be derived from the following set of rules:

Component Type Instances
$$C \vdash N(t) \Rightarrow |C(t)|$$

Total Provided Port Arity
$$C \vdash N(p) \Rightarrow \sum_{t \in \mathcal{U}\mathrm{P}(p)} |C(t)| \times \mathcal{U}(t).\mathbf{P}(p)$$

True
$$C \vdash \mathbf{true}$$

Comparison
$$\frac{C \vdash e \Rightarrow n' \quad n' \geq n}{C \vdash e \geq n}$$

Conjunction
$$\frac{C \vdash S_1 \quad C \vdash S_2}{C \vdash S_1 \wedge S_2}$$

We should notice, that in order to judge if a certain configuration satisfies a given specification, we need the context of a universe. This is caused by the fact, that in the specification we can ask for certain amount of a given port arity provided throughout the configuration. And the information linking the components with their provided port arity (i.e. the component type definition) is located in the universe. Without this feature we would not need a universe to decide if a configuration satisfies a specification.

5.2.6 Optimization

As we can see, the architecture synthesis problem is obviously not defined in a way that always pinpoints exactly a single right solution. Instead, in function of particular inputs (initial configuration, universe and specification) we can expect that there is either none, one or many final configurations that we would qualify as desired (i.e. which are valid solutions of this particular problem instance).

Infinity of solutions

In fact we should expect, that in most situations the number of valid solutions will be indeed infinite. The reason for this lies in the very structure of the architecture synthesis problem as we defined it, which does not contain any elements naturally imposing an upper bound on the size of the final configuration.

The constraints placed on the final configuration come from two sources: the universe and the specification. The universe restraints the repertoire of available component types and introduces functional and non-functional requirements on instances of these components, implemented as their ports and the arities associated with them. The specification, by the means of imposing conditions on the minimal number of certain components and on the total provided ports arity, establishes effectively a partial configuration that our final configuration must at least contain. Intuitively we can see it in the following way:

- from one side we constraint *the shape* of the final configuration by enforcing some regularities on it (encoded in form of the provided universe),
- from the other side we set its *minimal size* by posing a partial configuration (encoded in form of the provided specification).

As we can see, there is no explicit constraint concerning the *maximal size* of the final configuration.

In fact the only reason that it might have an upper size limit is if it is impossible to scale it up while staying valid with respect to the universe (i.e. intuitively if the *shape* constraints somehow the *maximal size*). However, although it is easy to imagine how these implicit size restrictions coming from the universe could work (e.g. through introducing singleton components using conflict ports), in most practical cases we should not expect them to be an issue, simply because modern distributed systems are usually deliberately designed to be scalable. Their size limits are mostly related to more mundane factors, like the finite available computing resources, and not to some inherent flaws in their design.

Thus, we should suppose that, as we stay on the high-level Aeolus model view of the distributed systems, which does not take into account such kind of constraints, we will often naturally encounter instances of the architecture synthesis problem that have an infinite number of solutions.

Choosing a solution

Either way, even in the cases where the number of acceptable final configurations is not infinite, we need to consider what are our options when we are being offered multiple possible solutions. In other words: how do we choose one of the available final configurations if all of them qualify as valid and desired and we must decide between them?

One way of approaching the problem is simply to assume that all the solutions which are acceptable are *equally good* and thus it does not matter which of them do we choose. However, a much more interesting option is to try to *evaluate* the solutions: establish some criteria of measure, that might help us decide which solutions are better or worse (i.e. less or more desirable in some way). At this point we are not thinking about how can we measure and compare an infinite number of solutions in practice, we simply want to imagine how could we compare one solution with another to identify the more interesting one.

Solution metrics

Number of components The most natural and simple way to measure a certain configuration which comes to mind is to count the number of the components it contains.

Most often when we have two solutions that are both correct and do the job, we will prefer to choose the cheaper one over the more expensive one. And the number of components is a fairly good estimate of the general amount of resources (e.g. computing power) which a certain configuration needs in order to be deployed and function properly.

Moreover, introducing such a criterion of choice gives us a mechanism that we can use to force the size of our final configurations to be reasonable and thus counterbalance the mentioned lack of other natural size-reducing factors in the architecture synthesis problem.

Cost of components Of course the number of components is not a perfect assessment of the cost of a configuration, as various types of components may have very different needs for resources. In order to refine this measure we could therefore attribute a cost value to every component type and calculate the total summary cost of all the components present in the configuration.

However we should bear in mind that the cost of deploying a certain distributed system in real world is in fact not linked directly to the deployed components, but rather to the cost of the machines that we deploy them on. And unfortunately for now we have no means to estimate this cost accurately: we do not know anything neither about the underlying architecture (be it virtual or real machines) nor about how our components will be deployed on it, because our model abstracts over this kind of issues.

Hence, in the context of assessing the actual expenses associated with deploying the whole system, attributing different costs to our component types does not seem to be a considerably more precise option than simply counting them.

Number of changes We can also imagine a completely different approach to measuring the final configuration: instead of establishing an *absolute* metric (like the two presented just before) and judging the final configuration on its own, we could design a *relative* metric and evaluate the final configuration in context of the initial one.

One of the most natural ideas for such a metric would be to assess the number of differences between the initial configuration and a given final one. This way we are not estimating the absolute deployment cost of the solution, but rather its relative distance from the chosen starting point. The question is: how do we exactly measure this distance or, in other words, the number of differences between the two configurations?

The easiest method which comes to mind is to simply compare the components present in both configurations. Each component that has been added, removed or that has changed its type in the meantime (which might correspond to it changing its state in the stateful model[2]) counts for one difference. A more precise comparison would additionally include assessing how the set of bindings has changed, but a measure based solely on components will be completely sufficient for our purpose here.

There is also another interesting question that arises: if we take into account this kind of metric when deciding among available solutions, then why should we not always choose a final configuration identical to the initial one? Although it is indeed the most natural option in many situations, one key condition must be fulfilled for it to be possible: the initial configuration must be an acceptable solution of the given problem instance. And, as we shall recall, the initial configuration of the architecture synthesis problem does not necessarily have to be valid with respect to the input universe and specification (as the final configuration is obliged to). Therefore posing such a metric seems completely reasonable, especially when there is doubt if the initial configuration is not a viable choice for the solution.

A practical example of such a situation would be when we want to adapt an already deployed system to new expectations (e.g. scale it up), while disturbing its current organization as little as possible. The initial configuration corresponds then to the current version of our system, but the universe and specification correspond to the desired new version (e.g. the redundancy requirements are stringier), therefore it is very probable that the initial configuration is not a valid solution of a problem defined in such a way.

Defining an optimization function

In order to establish a systematic approach defining how do we choose between multiple available solutions we will introduce a new concept: the *optimization function*[3].

What we call an *optimization function*[3]is any function that takes as an argument a configuration and returns a value indicating the utility of the given configuration in context of a certain optimization criterion. The co-domain of an optimization function may be any set of values (e.g. integer) with an associated *utility order*[3], which must be a total order.

When we want to choose the most desirable among a certain set of configurations, we try to find the one which returns the best value for our optimization function according to the utility order. In case of of an

[2]If we assume that names of components are preserved between the two configurations and that no deleted component name is reused, then indeed a component changing its type always corresponds to it changing its state in the stateful Aeolus model.

[3]These terms has been partly inspired by the mathematical optimization terminology in [109], but adapted to this particular context: our *optimization function* is similar to the *objective function* and our *utility order* is similar to the *sense of optimization*.

optimization function returning integers, it will usually simply be the *minimal* or *maximal* attainable value, depending on the desired criterion. Of course there may be multiple configurations which return the value corresponding to the highest utility, which means that all of them should be considered optimal solutions.

Compact If we take the "number of components in the configuration" criterion, the corresponding optimization function will simply return the number of components of a given configuration C. We call this one the *compact* optimization function:

$$compact(C) = |C_c|$$

As our goal will be to `minimize` this number, thus the utility order $\leq_{compact}$ that we define on the values of this function (which are natural numbers) is the reverse of the natural ordering:

$$\forall_{a,b \in \mathbb{N}} \qquad a \geq_{compact} b \quad \iff \quad a \leq b$$

Conservative If we take the criterion associated with our relative metric "distance from the initial configuration", the corresponding optimization will also need to take into account both the initial configuration $C_{init} = \langle W_{init}, B_{init} \rangle$ and final configuration $C_{fin} = \langle W_{fin}, B_{fin} \rangle$ and return the number of differences between them.

We count the differences in the following way:

- we take each component name $c \in C$ and if the component with this name is either:
  - present only in the initial configuration, (case 1)
  - or present only in the final configuration, (case 2)
  - or is present in both configurations, but changes its type, (case 3)

  we count it for one difference;
- components which:
  - are absent from both configurations
  - or are present in both, but have the same type,

  obviously do not count as a difference. (case 4)

We call it the *conservative* optimization function and our optimization goal is again to `minimize` its value:

$$conservative(C_{fin}) = \sum_{c \in C} \begin{cases} 1 & if \quad c \in dom(W_{init}) \wedge c \notin dom(W_{fin}) & \text{(case 1)} \\ 1 & if \quad c \notin dom(W_{init}) \wedge c \in dom(W_{fin}) & \text{(case 2)} \\ 1 & if \quad c \in dom(W_{init}) \wedge c \in dom(W_{fin}) \wedge W_{init}(c) \neq W_{fin}(c) & \text{(case 3)} \\ 0 & otherwise & \text{(case 4)} \end{cases}$$

$$\forall_{a,b \in \mathbb{N}} \qquad a \geq_{conservative} b \quad \iff \quad a \leq b$$

Formalization

The aim of the optimization function is to let us compare different solutions of an architecture synthesis problem instance according to some criterion in order to help us choose the best one. It does not serve as a way to assess the absolute value of a given final configuration, but rather it defines an order between the valid final configurations which makes sense only in a particular context (i.e. the architecture synthesis problem inputs: universe, initial configuration and specification).

Definition 15 (Optimization criteria)
We can define some optimization criteria *by providing:*

- *an* optimization function $f_{opt} : \Theta \to V_{opt}$ *which can be any function that takes a configuration as an argument,*
- *together with an* utility order $\left(V_{opt}, \leq_{opt}\right)$ *which must be a total order on the co-domain of* f_{opt}*.*

These two induce a total preorder[4] $\left(\Theta, \leq_{opt}\right)$ *on configurations configurations in the following way:*

$$\forall_{C^A_{fin}, C^B_{fin} \in \Theta} \quad C^A_{fin} \leq_{opt} C^B_{fin} \iff f_{opt}\left(C^A_{fin}\right) \leq_{opt} f_{opt}\left(C^B_{fin}\right)$$

This preorder represents the relative desirability / utility / quality of the configurations according to the given optimization criteria*, and*

$$C^A_{fin} \leq_{opt} C^B_{fin}$$

should be understood as:

"configuration C^B_{fin} *is not worse than configuration* C^A_{fin}*"*

For completeness, let us define the two most obvious utility orders on integers (i.e. $V_{opt} = \mathbb{Z}$):

- we use the natural integer ordering as our utility order if we want to `maximize` an optimization function of form $f_{opt} : \Theta \to \mathbb{Z}$ (i.e. when higher values correspond to better final configurations):

$$\forall a, b \in \mathbb{Z} \qquad a \leq_{opt} b \qquad \iff \qquad a \leq b$$

- and we use the reverse of the natural integer ordering as our utility order if we want to `minimize` an optimization function of form $f_{opt} : \Theta \to \mathbb{Z}$ (i.e. when lower values correspond to better final configurations):

$$\forall a, b \in \mathbb{Z} \qquad a \leq_{opt} b \qquad \iff \qquad b \leq a$$

5.2.7 Conclusion

In this section we have set up the whole formal framework for precisely stating the architecture synthesis problem and describing all its elements. First we will define the problem itself and then separately the optimal solution of the problem:

Definition 16 (Architecture synthesis problem)
Given

- *an initial configuration* C_{init}*,*
- *a universe* $\mathcal{U}$
- *and a specification* S*,*

find a final configuration C *such that*

- C *is valid w.r.t.* $\mathcal{U}$ *(see definition 13 on page 73)*
- *and* C *satisfies* S *(see definition 14 on page 74).*

Definition 17 (Optimal solution of the architecture synthesis problem)
Given architecture synthesis problem inputs:

[4] It is not a *total order* any more, because two configurations C_A and C_A can be different, yet correspond to the same utility and thus $C_A \leq_{opt} C_B \wedge C_B \leq_{opt} C_A$ and still $C_A \neq C_B$.

- *an initial configuration* C_{init},
- *a universe* $\mathcal{U}$,
- *a specification* S

and given formalized optimization criteria *(see definition 15 on page 77):*

- *an optimization function* $f_{opt} : \Theta \to V_{opt}$,
- *with an utility order* $\left(V_{opt}, \leq_{opt}\right)$

which together induce a total preorder on configurations $\left(\Theta, \leq_{opt}\right)$,

an optimal solution *of this architecture synthesis problem according to the provided optimization criteria is a final configuration* C_{fin}^{opt} *which is every maximal element among all the correct solutions of this problem in the preorder* $\leq_{opt}$*, i.e. for each final configuration* C_{fin} *which is a correct solution of the problem (see definition 16 on page 78),* C_{fin}^{opt} *must be* not worse *than* C_{fin} *according to the preorder* $\leq_{opt}$*:*

$$C_{fin} \leq_{opt} C_{fin}^{opt}$$

In the next section we will discuss how the architecture synthesis problem, both without and with optimization criteria, can be solved in practice.

5.3 Constraint solving approach

We have by now well defined the architecture synthesis problem and its elements. We have also established all the formal foundations necessary to approach it in a strict way: all the definitions and notations have been introduced. Now the time has come to tackle the challenge of solving the problem in practice.

5.3.1 Search for solutions

Unfortunately it is not easy to find a good and fast algorithm that we could use to solve efficiently any instance of the architecture synthesis problem. Although the problem itself is considerably less complicated than the original one that we wanted to solve (i.e. the reconfigurability problem defined in chapter 4, which involves not only finding a desired and valid final configuration, but also a deployment plan to reach it from the initial configuration), it still remains rather difficult to tackle.

Diversity of instances

The instances of the architecture synthesis problem are potentially very different from one another. Because of that no single simplex approach seems to be universally efficient in dealing with all of them. The reason for this comes from the complexity of the stateless Aeolus model, which is indeed quite expressive. As we use the model to define a big part of the architecture synthesis problem's inputs (i.e. the initial configuration and the universe), they are in turn quite complex entities, which may take various very different shapes.

Adopting the constraint solving approach

We tried for some time to come up with an efficient way to solve instances of the architecture synthesis problem in practice. At the beginning we have considered several different direct approaches (like designing a dedicated algorithm), but most of our ideas were not really satisfying. We have reached quickly a conclusion that, by the cause of the complex structure of the problem and the highly heterogeneous nature of its instances, such techniques are not very well suited for solving it in general and we have decided that

the best course of action would be to employ some kind of a more flexible and adjustable method. This is how we came up with the idea of using the constraint solving approach.

So far it is the only technique that has proved itself to be conveniently applicable to all the instances of the architecture synthesis problem and moreover to be usually reasonably efficient in practice. Is is indeed the best option we have discovered yet.

Advantages of constraint solving

There are many reasons why the constraint solving approach seems to be rather well adapted to the task at hand in this case:

- First of all, the declarative nature of constraint solving makes it particularly apt to handle this kind of complex problems with highly diverse and heterogeneous instances. We only specify the problem which needs to be solved instead of specifying directly *how* it should be solved and, in some sense, constraint solving does inherently "adapt" to the individual instances of the problem, as the manner in which it traverses the solution space may depend on the shape of a particular problem instance.

  Also, capturing the architecture synthesis problem in form of a set of constraints was maybe not exactly trivial, but it certainly proved itself to be much easier than finding a specific algorithm that would solve such problems in a satisfying way. This is especially true if we want to find optimal solutions, as designing an efficient algorithm capable of finding an optimal solution is an even more difficult challenge, while constraint solving supports optimization intrinsically.

- For every flavour of constraint solving (linear programming, integer domains, SAT, etc.) there are numerous state of the art solvers available on the market, many of which are FOSS. This gives us a wide array of possible solver components to choose from when implementing this solution in practice.

- Further adaptation of the solving process is possible:

  - many constraint solvers use sophisticated methods of adapting their search strategy in function of each given problems.
  - also many provide us a way to manually tune the search strategy using our higher-level knowledge about the particular problem's structure.

- Another important advantage of the constraint solving approach is the considerable degree of flexibility it exhibits with regard to the changes in the underlying problem.

  This happened to be a very valuable property, especially when we were tweaking the definition of the architecture synthesis problem and altering the structure details of the stateless Aeolus model. In fact it was quite easy and natural to adjust the corresponding constraint problems accordingly instead of redesigning them from scratch. It would have been probably much harder if we have used a more direct approach and, for example, needed to reshape a fixed algorithm.

5.3.2 The basic idea

In order to be able to solve the architecture synthesis problem using constraint solving approach we need two ingredients:

1. we have to be able to convert any *instance of our original problem* into an equivalent *constraint problem instance*, expressed in form of variables and constraints on these variables

2. and we have to be able to convert the *solution of this constraint problem*, expressed in form of a mapping from variables to values, into *a solution of our original problem*.

An architecture synthesis problem instance is defined by an initial configuration, a universe, a request (specification) and optionally an optimization function. The solution we are looking for is a final configuration that is valid with respect to the universe, satisfies the request and optionally minimizes or maximizes the optimization function.

The two mentioned conversions constitute the backbone of our method, as they allow us to find solutions of given architecture synthesis problem instances indirectly, by solving corresponding constraint problems instead.

Constraint problem

Before getting into further details, let us take some time to describe more clearly what do we mean exactly when we talk about *a constraint problem* in this context (for a similar approach to the one presented here see for example [9]).

Basics A constraint problem essentially consists of two elements:

- a set of *variables*, each over a defined *domain*,
- and a set of *constraints* stating relations between these variables.

Solving a constraint problem involves attributing to every variable a single value from its domain in a way that respects the constraints. Therefore a solution of a constraint problem which contains a given set of variables is a function from these variables to their values.

If we define an order on the solutions, then the optimal solution of a constrain problem is the one that is the minimal or maximal one (depending on the optimization goal) in this order.

Variation There exist many different variations of constraint systems (a survey of some of them can be found in [66]), defined by the type of the variable domains used and the available constraints repertoire. For example, one of very important variants, related to the boolean satisfiability problem (SAT) [23], restricts the variable domains to boolean values and constraints to propositional logic formulas over the variables.

In the current scope we will be using only variables over integer domains (i.e. in $\mathbb{Z}$) and our repertoire of available constraints will be well defined and limited to a small subset of the most simple ones typically used in finite domain constraint solvers. Thanks to that it will be easy to apply our approach in practice, as the constraint problems that we consider here theoretically are in fact almost directly encodable in most standard constraint description languages for many state of the art finite domain solvers.

C	::=	$\mathbf{true} \mid e\ op\ e \mid C \wedge C \mid C \vee C \mid C \Rightarrow C \mid \neg C$	Constraint
e	::=	$n \mid v \mid e + e \mid e - e \mid n \times e \mid \lvert e \rvert \mid \lVert C \rVert$	Expression
op	::=	$< \mid \leq \mid = \mid \geq \mid > \mid \neq$	Comparison
opt	::=	$\max(e) \mid \min(e)$	Optimization Criterion

Table 5.1: Constraint syntax and constraint problem optimization criteria syntax

Constraint syntax Table 5.1 presents the formal syntax of our constraints. Basically:

- a **constraint** C is a set of comparisons between numerical expressions $e\ op\ e$, combined using the logical operators $\wedge$, $\vee$, $\Rightarrow$ and $\neg$;
- an **expression** e is simply a numerical expression, with integers n, variables v, sum, subtraction, multiplication (multiplication works only with a constant, we cannot multiply two variables!) and absolute value, extended with reified constraints $\lVert C \rVert$, whose value is 1 if C is true, 0 otherwise;
- an **optimization criterion** *opt* is either to `maximize` or to `maximize` the expression e (it is not a part of a constraint, but a way to choose among a constraint's solutions).

$$\textsc{True}\quad \sigma \Vdash \textbf{true}$$

$$\textsc{Comparison}\quad \frac{\sigma \Vdash e_1 \Rightarrow n_1 \qquad \sigma \Vdash e_2 \Rightarrow n_2 \qquad n_1\ op\ n_2}{\sigma \Vdash e_1\ op\ e_2}$$

$$\textsc{Conjunction}\quad \frac{\sigma \Vdash C_1 \qquad \sigma \Vdash C_2}{\sigma \Vdash C_1 \wedge C_2}$$

$$\textsc{Disjunction 1}\quad \frac{\sigma \Vdash C_1}{\sigma \Vdash C_1 \vee C_2}$$

$$\textsc{Disjunction 2}\quad \frac{\sigma \Vdash C_2}{\sigma \Vdash C_1 \vee C_2}$$

$$\textsc{Implication 1}\quad \frac{\sigma \Vdash C_1 \qquad \sigma \Vdash C_2}{\sigma \Vdash C_1 \Rightarrow C_2}$$

$$\textsc{Implication 2}\quad \frac{\sigma \nVdash C_1}{\sigma \Vdash C_1 \Rightarrow C_2}$$

$$\textsc{Negation}\quad \frac{\sigma \nVdash C}{\sigma \Vdash \neg C}$$

$$\textsc{Constant}\quad \sigma \Vdash n \Rightarrow n$$

$$\textsc{Variable}\quad \sigma \Vdash v \Rightarrow \sigma(v)$$

$$\textsc{Sum}\quad \frac{\sigma \Vdash e_1 \Rightarrow n_1 \quad \sigma \Vdash e_2 \Rightarrow n_2}{\sigma \Vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

$$\textsc{Subtraction}\quad \frac{\sigma \Vdash e_1 \Rightarrow n_1 \quad \sigma \Vdash e_2 \Rightarrow n_2}{\sigma \Vdash e_1 - e_2 \Rightarrow n_1 - n_2}$$

$$\textsc{Multiplication}\quad \frac{\sigma \Vdash e \Rightarrow n'}{\sigma \Vdash e \times n \Rightarrow n' \times n}$$

$$\textsc{Reification 1}\quad \frac{\sigma \Vdash C}{\sigma \Vdash ||C|| \Rightarrow 1}$$

$$\textsc{Reification 2}\quad \frac{\sigma \nVdash C}{\sigma \Vdash ||C|| \Rightarrow 0}$$

Table 5.2: Constraint semantics

Constraint semantics The semantics of our constraints is the natural one: a solution σ for a constraint C is a mapping from variables to their values, such that substituting the variables by their values in C will result in a true statement. For completeness, we present in table 5.2 the full semantics of each constraint. We note $\sigma \Vdash C$ when the variable mapping σ is a solution for C.

An optimal solution σ_{opt} for a constraint C is a solution for C which is minimizing or maximizing (depending on the optimization criterion) given expression e.

Definitions Let us now define the constraint problem formally, as we will use it in our approach:

Definition 18 (Constraint problem)
A constraint problem P *is a triple* $\langle \mathcal{V}, \mathcal{D}, C \rangle$*, consisting of:*

- *a set of variables* $\mathcal{V}$*;*
- *a function* $\mathcal{D} : \mathcal{V} \to \wp(\mathbb{Z})$*, from variables to their domains (which are subsets of integer numbers);*
- *a constraint* C*;*

where every variable that is mentioned in C *must belong to* $\mathcal{V}$*.*

For notation simplicity instead of putting in the constraint problem definition a whole set of constraints to be satisfied we only have a single constraint C per problem. As a set of constraints that all need to be fulfilled is obviously completely equivalent to fulfilling a single constraint which is a conjunction of all the constraints from such a set, these two formulations are interchangeable.

Definition 19 (Constraint problem solution)
Given a constraint problem $\mathrm{P} = \langle \mathcal{V}, \mathcal{D}, C \rangle$*, a solution* $\sigma : \mathcal{V} \mapsto \mathbb{Z}$ *for such a problem is a mapping from the variables to values, such that:*

- *the value of each variable belongs to this variable's domain:* $\bigwedge_{v \in \mathcal{V}} \sigma(v) \in \mathcal{D}(v)$,
- *and substituting the variables by their values in* C *results in a correct solution (according to the constraint semantics presented in the table 5.2):* $\sigma \Vdash C$

We note $\sigma \Vdash \mathrm{P}$ *when the variable mapping* σ *is a solution for the constraint problem* P.

Definition 20 (Minimum and maximum optimization criteria utility)
A optimization criterion *opt induces the* utility preorder $\leq_{opt}$ *on solutions (i.e. variable mappings) in the following way:*

- *if the optimization criterion opt is* $\max(e)$*, then solutions in which the value of expression e is higher are considered better,*
- *if the optimization criterion opt is* $\min(e)$*, then solutions in which the value of expression e is lower are considered better.*

$$\forall_{\sigma_1,\sigma_2} \begin{cases} \sigma_1 \leq_{opt} \sigma_2 \iff \exists n_{\sigma_1}, n_{\sigma_2} \begin{cases} \sigma_1 \Vdash e \Rightarrow n_{\sigma_1} \\ \sigma_2 \Vdash e \Rightarrow n_{\sigma_2} \\ n_{\sigma_1} \leq n_{\sigma_2} \end{cases} & \text{if} \quad opt = \max(e) \\ \\ \sigma_1 \leq_{opt} \sigma_2 \iff \exists n_{\sigma_1}, n_{\sigma_2} \begin{cases} \sigma_1 \Vdash e \Rightarrow n_{\sigma_1} \\ \sigma_2 \Vdash e \Rightarrow n_{\sigma_2} \\ n_{\sigma_1} \geq n_{\sigma_2} \end{cases} & \text{if} \quad opt = \min(e) \end{cases}$$

Definition 21 (Constraint problem optimal solution)
Given

- *a constraint problem* $\mathrm{P} = \langle \mathcal{V}, \mathcal{D}, C \rangle$
- *and an* optimization criterion *opt*

an optimal solution σ_{opt} *of problem* P *according to the optimization criterion opt is*

- *each correct solution of the constraint problem* P *(see definition 19)*
- *which is also the* maximal element *among all the solutions of problem* P *in the utility preorder* $\leq_{opt}$ *corresponding to the optimization criterion opt:*

$$\sigma_{opt} \text{ is an optimal solution of } \mathrm{P} \iff \begin{cases} \sigma_{opt} \Vdash \mathrm{P} \\ \forall_{\sigma} \quad \left(\sigma \Vdash \mathrm{P} \ \wedge \ \sigma_{opt} \leq_{opt} \sigma\right) \ \rightarrow \ \sigma_{opt} = \sigma \end{cases}$$

Schema of action

Our approach consists of three steps, which take us from the *original problem level* (expressed in the Aeolus stateless model form) to the *constraint problem level* and back:

1. first we encode the original problem (i.e. an initial configuration, a universe, a request, and optionally an optimization criterion) into a constraint problem form,
2. then we use an adapted constraint solver to find a solution of this generated constraint problem,
3. finally we translate the constraint problem's solution back to the form of the original problem's solution (i.e. a final configuration), thus attaining our goal.

The whole process is illustrated on figure 5.2.

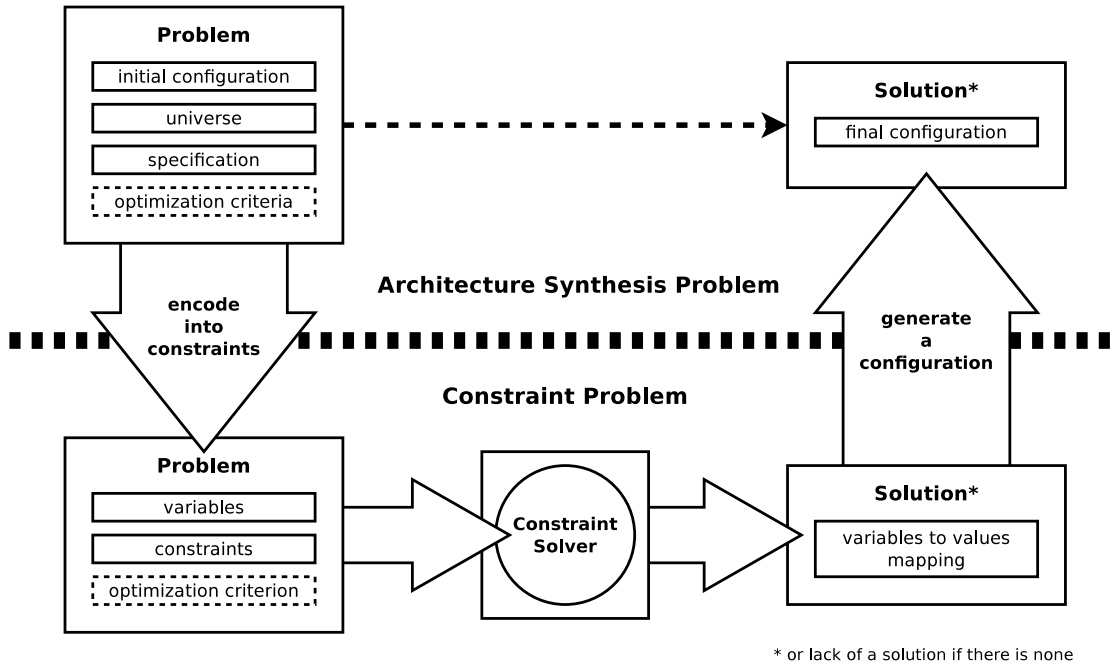


Figure 5.2: Solving architecture synthesis problems by passing through the constraint problems domain.

5.3.3 Encoding

Let us see now how do we implement the two conversions that allow us to use the constraint solving approach in solving the architecture synthesis problem.The principal idea here is to encode the desired final configuration in form of a certain (finite) number of named variables and capture the requirements concerning it, which come from all the parts of the problem's input, in form of constraints imposed on these variables.

We will use the integer constraint solving and our variables will be non-negative integers, their domains ranging by default from zero to infinity (i.e. natural numbers with zero). Although on the theoretical level this kind of infinite domains are completely acceptable, we should bear in mind that in practice we will rather switch to the much more pragmatic and efficient finite domain approach[5] (where variables must be defined over finite sets): we will simply specify a sufficiently big number as the upper bound of our variable domains (i.e. a pseudo-infinity).

Of course domains of all the variables can always be adjusted more precisely, for example some of them can be boolean (i.e. take only values zero or one).

Main idea

The main issue that we encounter when we try to encode a final configuration in form of a certain number of named variables is that we do not know beforehand how many components and bindings will that configuration contain. Therefore we cannot, for example, simply attribute a single boolean variable to each component, where zero would correspond to absence of this component in the configuration and one to its presence.

However, we can profit from the fact that the final configuration always has to be valid with respect to the universe provided in the inputs of the problem (which is hence known to us). And the first basic condition of such validity is that all the components present in the configuration should be instances of component types available in this universe. Consequently, we can treat the components present in the

[5]Finite domain constraint programming has considerable practical applications and is one of the most developed domains of constraint programming. There are many very efficient and free state-of-the-art finite domain constraint solvers available which we can potentially use in the implementation of our approach.

configuration in a collective manner and, instead of attributing a single boolean variable to each component, we can attribute a single integer variable to each component type: the variable's value corresponds to the number of the components of the given type present in the configuration. As we know all the available component types beforehand, doing this will always be possible.

This way of encoding the final configuration has unfortunately one important disadvantage: as we represent the components *en masse*, we lose altogether the possibility to differentiate them and thus give each a separate identity. Because of that, as we cannot refer to a certain component, but solely to entire groups of them (i.e. all the instances of a given component type), all the other aspects of the configuration and its validity, namely the bindings between components and all the requirements concerning them, necessarily also have to be encoded respecting this aggregate style. In result, instead of representing separately each binding between two specific components (which in fact either way would not be feasible because their number is not known beforehand), we can only count how many bindings on a particular port are formed in total between two given groups of components (i.e. from providing components of a specific type to the requiring components of another specific type).

Variables

All that gives us two sets of variables with which we encode the final configuration of a given instance of the architecture synthesis problem:

- The **component type** variables: $N(t)$

  A component type variable corresponding to a component type with name t is $N(t)$.

  The value of that variable represents how many instances of this component type are present in the final configuration.

- The **binding** variables: $B(p, t_r, t_p)$

  A binding variable $B(p, t_r, t_p)$ corresponds to the bindings which:

  - are on port p,
  - the name of the type of their requiring component is t_r,
  - the name of the type of their providing component is t_p.

  The value of that variable represents how many bindings of this kind exist in the configuration (i.e. each binding is characterized here by these three properties and all the bindings with the three matching properties are treated collectively).

In order to represent any potential final configuration of an architecture synthesis problem where the input universe is $\mathcal{U}$ using this method we create a constraint problem which includes at least the following variables:

- A non-negative integer variable $N(t)$ for each component type name defined in the universe $t \in \mathcal{U}_{dt}$;
- A non-negative integer variable $B(p, t_r, t_p)$ for each kind of bindings that can potentially exist between the instances of the component types available in the universe. Which means more precisely, all the possible triples consisting of: a port used in the universe $p \in \mathcal{U}_{dp}$, a component type requiring that port $t_r \in \mathcal{U}\mathrm{R}(p)$ and a component type providing that port $t_p \in \mathcal{U}\mathrm{R}(p)$.

Let us define formally the set of variables $\mathcal{V}$ and the function attributing them their domains $\mathcal{D}$ of a constraint problem $\mathrm{P} = \langle \mathcal{V}, \mathcal{D}, C \rangle$, corresponding to an architecture synthesis problem with an input universe $\mathcal{U}$:

$$\mathcal{V} = \bigcup \begin{cases} \bigcup\limits_{t \in \mathcal{U}_{dt}} N(t) \\ \bigcup\limits_{p \in \mathcal{U}_{dp}} \quad \bigcup\limits_{t_r \in \mathcal{U}\mathrm{R}(p)} \quad \bigcup\limits_{t_p \in \mathcal{U}\mathrm{R}(p)} B(p, t_r, t_p) \end{cases}$$

$$\forall v \in \mathcal{V}, \quad \mathcal{D}(v) = \mathbb{N}$$

Superfluous variables We have defined the set of all the variables which are necessary in order to represent any final configuration that could interest us. However, we could easily add here more component type and binding variables, corresponding to component types which are not available in the given universe or bindings that cannot be created in the final configuration (e.g. between the component types that do not require nor provide a given port).

For example, we could include the variables corresponding to the *deprecated* component types: those which are not available in the universe, but some instances of them are present the initial configuration (thus we can imagine that they were available in the universe used previously).

Our constraint problem will make perfect sense even with such superfluous variables. However, the values assigned to of all of them will obviously have to be equal to zero, as the components and bindings represented by them must not be present in the solution.

Constraints

Obviously not all possible combinations of values of a given constraint problem's variables will lead to correct solutions of the corresponding architecture synthesis problem instance.

In order for a final configuration to be a correct solution, we must make sure that it is valid with respect to the input universe $\mathcal{U}$ and that it satisfies the input specification S. We must thus introduce such constraints on our variables that will correspond to all the rules by which the equivalent final configuration has to abide.

First: universe validity This requires us to reformulate the universe validity constraints that we have introduced before (more exactly in the subsection 5.2.5) in order to make them work well with the collective fashion of representing the configuration's components and bindings, unavoidable in the constraint problem translation of the architecture synthesis problem. Therefore we must transform the rules that were defined in terms of individual components and bindings into a form based on component types and numbers of bindings between whole groups of components (which corresponds to our variables).

- First rule: **requires must be satisfied**

  When approaching components individually, we say, that each require port of a given component must be bound to matching provide ports of a number of different components, that is at least as big as its require arity.

  In the collective approach we can only introduce a similar constraint referring to whole groups of components: all the instances of a given component type. As in each such group all the components are exactly identical, we can simply count the total require arity that they demand together on a certain port (which equals number of components multiplied by the port's require arity) and compare it with the number of bindings that they are provided on that port:

  $$\bigwedge_{p \in \mathcal{U}_{dp}} \bigwedge_{t_r \in \mathcal{U}\mathrm{R}(p)} \mathcal{U}(t_r).\mathbf{R}(p) \times N(t_r) \quad \leq \sum_{t_p \in \mathcal{U}\mathrm{P}(p)} B(p, t_p, t_r)$$

  We should note, that these constraints alone do not assure that each requiring component will be bound to enough *different* providing components. They only count the total values corresponding to the amount of bindings that is required by each group of components, and this is not sufficient to guarantee that the bindings in question can be established in a way that respects all the appropriate rules. We will fix this problem shortly.

- Second rule: **provides must not be overused**

  This rule works similarly to the previous one, but focuses on the provide ports. The idea is to limit the number of existing bindings providing a given port depending on the number of components that actually provide this port.

  In this rule however, as the port's provide arity can either be finite or infinite, we have to take into account these two possibilities and create a separate constraint for both of them:

– If a certain component type's provide arity on a given port p is finite, then everything will be almost exactly symmetrical to the previous rule:

$$\bigwedge_{p \in \mathcal{U}_{dp}} \bigwedge_{t_p \in \{t \in \mathcal{U}\mathrm{P}(p) \mid \mathcal{U}(t).\mathbf{P}(p) < \infty\}} \mathcal{U}(t_p).\mathbf{P}(p) \times N(t_p) \quad \geq \sum_{t_r \in \mathcal{U}\mathrm{R}(p)} B(p, t_p, t_r)$$

– On the other hand, if the component type's provide arity on port p is infinite and at least one component of this type is present, then the number of available bindings on port p is effectively unbounded. Thus we will only need to constraint the number of the corresponding bindings if there are none such components present:

$$\bigwedge_{p \in \mathcal{U}_{dp}} \bigwedge_{t_p \in \{t \in \mathcal{U}\mathrm{P}(p) \mid \mathcal{U}(t).\mathbf{P}(p) = \infty\}} N(t_p) = 0 \quad \Rightarrow \sum_{t_r \in \mathcal{U}\mathrm{R}(p)} B(p, t_p, t_r) = 0$$

Exactly as in the previous case, these constraints on their own work only on the global level and are not sufficient to assure that each specific providing component's provide arity will not be overused.

- Complementing the first and second rule: **binding unicity**

  As mentioned before, the constraints introduced so far are not really capable of assuring that the two first rules of the configuration validity with respect to the universe (i.e. *requires must be satisfied* and *provides must not be overused*) will be fully obeyed in the final configuration.

  The constraints which concern the summary require arities ensure that there will be *in total* enough bindings to satisfy all the components' requires. Conversely, the constraints concerning the summary provide arities ensure that there will be *in total* enough providing components with enough provide arity to actually provide all these bindings. What we cannot hope to ensure on this level at all (i.e. with the collective representation of components and bindings), is that each *individual* requiring or providing component will get its fair share of the concerned require or provide bindings[6].

  However, there is another issue which can and should be addressed exactly on this level. As we know, the redundancy requirements state that all the bindings used by each requiring component should come from *different* providing components. Therefore, we need to assure that not only there is *enough* bindings for all the requiring components, but also that there is enough *different* bindings (i.e. bindings provided by different providing components) for all the requiring components. And (surprisingly) this kind of guarantee is entirely possible to attain in the collective context that we operate in right now.

  This exact property (i.e. each requiring component must be bound to enough different providing components) is equivalent to making sure, that no requiring component is bound twice to the same providing component. Which brings us straight to the *binding unicity principle*[7].

  The binding unicity principle is an important feature which is almost automatically present in the Aeolus model representations of the configuration and which we have lost when representing our components and bindings *en masse*. It is enough to enforce the bindings unicity principle on the solutions, in addition to the already introduced constraints, in order to fix our problem and really assure that our two validity rules hold (i.e. *requires must be satisfied* and *provides must not be overused*[8]).

  Attaining this aim becomes feasible after making one observation:

[6] As our constraints can operate only on global sums of provide or require arities and global amounts of different kinds of bindings, there is simply no possibility in this context to address the problem of distributing the bindings in a right way, i.e. basically giving *enough* to each single requiring component and not taking *too much* from any single providing component. This whole issue will became tractable only on the next level, when generating the actual final configuration, and is simply out of our scope for now.

[7] The rule, that there can be at most one binding from a given provided port to a given required port between two given components. Already mentioned multiple times in chapter 4 as well as in subsection 5.2.3 of the current Chapter.

[8] Modulo the aforementioned issue of distributing the bindings correctly among the individual components.

It is enough to restrict the number of bindings on a given port p between requiring components of a certain component type t_r and providing components of a certain component type t_p (i.e. $B(p, t_p, t_r)$) to the product of number of instances of these two component types (i.e. $N(t_r) \times N(t_p)$) in order to enforce the binding unicity principle.

If we represent our two groups of components as a bipartite graph, with the requiring components on one side and the providing ones on the other, it becomes evident, that the maximal number of edges in this graph is restrained by such an equation. In other words, if we consider similar bipartite multi-graphs (i.e. graphs allowing multiple edges between any two nodes), this would be a necessary condition to verify if a given bipartite multi-graph is also a normal graph (i.e. it contains no more than one edge between any two nodes). Therefore intuitively this constraint seems to correspond well to the binding unicity principle, where we want to check if each two components are connected with at most one binding (of each kind).

With the following constraints we are thus enforcing the binding unicity principle and consequently ensuring, that the bindings between require and provide ports in the final configuration will be valid[8]:

$$\bigwedge_{p \in \mathcal{U}_{dp}} \bigwedge_{t_r \in \mathcal{UR}(p)} \bigwedge_{t_p \in \mathcal{UP}(p)} B(p, t_p, t_r) \leq N(t_r) \times N(t_p)$$

- Third rule: **there must be no conflicts**

  The last important rule that the final configuration must follow is the absence of conflicts. If a certain component has a conflict port, then it can be the only component present in the configuration that provides this port. Because of the particular structure of this rule, in order to express it in form of constraints on our variables, we will divide it into three separate parts.

  First we will directly take care of the singleton components and make sure that every component type which is in conflict with itself (i.e. in the same time provides a port and is in conflict with it) will have at most one instance present in the final configuration (as if there were any more instances, they would be obviously in conflict with each other, so we can either have one or none):

  $$\bigwedge_{t \in \mathcal{UC}(p) \cap \mathcal{UP}(p)} N(t) \leq 1$$

  Next thing to do is to introduce constraints which take care of the conflicts between different component types. If even a single component that conflicts with port p is present in the configuration, then no other component that provides this port can exist. In order to keep everything completely clear, we will separate this into two constraints, one that deals explicitly with the self-conflicted component types (working in synergy with the recently established singleton-related constraint):

  $$\bigwedge_{t \in \mathcal{UC}(p) \cap \mathcal{UP}(p)} N(t) = 1 \quad \Rightarrow \bigwedge_{t' \in \mathcal{UP}(p), t' \neq t} N(t') = 0$$

  and a second one, that deals with all the other (i.e. not self-conflicted) component types:

  $$\bigwedge_{t \in \mathcal{UC}(p) \setminus \mathcal{UP}(p)} N(t) > 0 \quad \Rightarrow \bigwedge_{t' \in \mathcal{UP}(p)} N(t') = 0$$

  We can see that, as conflicts do not have anything to do directly with bindings, the conflict-related constraints are based solely on the component type variables and do not include any references to the binding variables.

- Supplementary rules: **impossible components and bindings**

  As mentioned before, nothing prevents us from putting more component type and binding variables in our constraint problem than is strictly necessary. If we want, we can add variables that correspond:

– to component types which do not exist in our input universe (e.g. the *deprecated* component types which have instances in the initial configuration, but are not available in the universe),
– or to bindings that cannot be established because the concerned providing component does not possess the matching provide port,
– or finally to bindings that technically can be established (as specified in our formal model), but are superfluous by definition: those where the concerned requiring component does not possess the matching require port.

Introducing such unnecessary variables does not harm us in any way and in some situations it may facilitate certain problem encodings in practice (like adding the variables for *deprecated* component types). Thus we want to keep this possible and, if variables of this kind are present in the variable set $\mathcal{V}$, we simply add some constraints that keep their values right (which means equal zero):

– We should make sure, that no component type from outside of our input universe is instantiated in the final configuration:

$$\bigwedge_{t \in \left\{ t \;\middle|\; \substack{N(t) \in \mathcal{V} \\ \wedge \\ t \notin \mathcal{U}_{dt}} \right\}} N(t) = 0$$

– Also we should make sure that no impossible binding actually exists: if a component does not provide the port p, it cannot be on the providing end of a binding concerning this port[9] :

$$\bigwedge_{(p,t_p,t_r) \in \left\{ (p,t_p,t_r) \;\middle|\; \substack{B(p,t_p,t_r) \in \mathcal{V} \\ \wedge \\ t_p \notin \mathcal{UP}(p)} \right\}} B(p, t_p, t_r) = 0$$

– On the other hand, if a component does not require the port p, it can still technically be on the requiring end of a binding concerning this port. However, such bindings do not interfere at all with our validity rules. Intuitively they correspond to "inactive" bindings in the stateful model (i.e. bindings with an inactive require port), thus they do not consume the provide arity of the providing component. Therefore we do not really need to introduce any constraints controlling them, apart from the ones needed to ensure binding unicity.

  Nevertheless, for completeness, let us define the constraints which we could introduce if we wanted to get rid of such useless bindings:

$$\bigwedge_{(p,t_p,t_r) \in \left\{ (p,t_p,t_r) \;\middle|\; \substack{B(p,t_p,t_r) \in \mathcal{V} \\ \wedge \\ t_r \notin \mathcal{UR}(p)} \right\}} B(p, t_p, t_r) = 0$$

In many cases, for practical reasons, we want to introduce explicitly the variables and constraints related to the mentioned *deprecated* component types (i.e. those which have instances in the initial configuration, but are not available in the universe). If we add the variables $N(t)$ concerning such component types to our set $\mathcal{V}$ then, applying the provided rules for the unnecessary variables, we will also extend our constraints with:

$$\bigwedge_{t \in C_t^{init} \setminus \mathcal{U}_{dt}} N(t) = 0$$

The whole encoding of the the core part of the configuration's validity with respect to a universe in form of a constraint is summarized in table 5.3. The supplementary part which takes care of the unnecessary variables is summarized in table 5.4.

[9] In the stateless model components have no longer internal state machines, so that only the active ports are present; hence bindings connect only active ports.

$$\bigwedge_{p\in\mathcal{U}_{dp}} \bigwedge_{t_r\in\mathcal{U}\mathrm{R}(p)} \mathcal{U}(t_r).\mathbf{R}(p) \times N(t_r) \leq \sum_{t_p\in\mathcal{U}\mathrm{P}(p)} B(p,t_p,t_r) \tag{5.1}$$

$$\bigwedge_{p\in\mathcal{U}_{dp}} \begin{cases} \bigwedge\limits_{t_p\in\{t\in\mathcal{U}\mathrm{P}(p)|\mathcal{U}(t).\mathbf{P}(p)<\infty\}} \mathcal{U}(t_p).\mathbf{P}(p) \times N(t_p) \geq \sum\limits_{t_r\in\mathcal{U}\mathrm{R}(p)} B(p,t_p,t_r) \\ \bigwedge\limits_{t_p\in\{t\in\mathcal{U}\mathrm{P}(p)|\mathcal{U}(t).\mathbf{P}(p)=\infty\}} N(t_p) = 0 \Rightarrow \sum\limits_{t_r\in\mathcal{U}\mathrm{R}(p)} B(p,t_p,t_r) = 0 \end{cases} \tag{5.2}$$

$$\bigwedge_{p\in\mathcal{U}_{dp}} \bigwedge_{t_r\in\mathcal{U}\mathrm{R}(p)} \bigwedge_{t_p\in\mathcal{U}\mathrm{P}(p)} B(p,t_p,t_r) \leq N(t_r) \times N(t_p) \tag{5.3}$$

$$\bigwedge_{p\in\mathcal{U}_{dp}} \begin{cases} \bigwedge\limits_{t\in\mathcal{U}\mathrm{C}(p)\cap\mathcal{U}\mathrm{P}(p)} N(t) \leq 1 \\ \bigwedge\limits_{t\in\mathcal{U}\mathrm{C}(p)\cap\mathcal{U}\mathrm{P}(p)} N(t) = 1 \Rightarrow \bigwedge\limits_{t'\in\mathcal{U}\mathrm{P}(p),t'\neq t} N(t') = 0 \\ \bigwedge\limits_{t\in\mathcal{U}\mathrm{C}(p)\setminus\mathcal{U}\mathrm{P}(p)} N(t) > 0 \Rightarrow \bigwedge\limits_{t'\in\mathcal{U}\mathrm{P}(p)} N(t') = 0 \end{cases} \tag{5.4}$$

Table 5.3: Summary of the translation of the core rules concerning the configuration validity w.r.t. universe $\mathcal{U}$ into a constraint (without the supplementary rules for superfluous variables).

$$\bigwedge_{t\in\left\{t \,\middle|\, \substack{N(t)\in\mathcal{V} \\ \wedge \\ t\notin\mathcal{U}_{dt}}\right\}} N(t) = 0 \tag{5.5}$$

$$\bigwedge_{(p,t_p,t_r)\in\left\{(p,t_p,t_r) \,\middle|\, \substack{B(p,t_p,t_r)\in\mathcal{V} \\ \wedge \\ t_p\notin\mathcal{U}\mathrm{P}(p)}\right\}} B(p,t_p,t_r) = 0 \tag{5.6}$$

$$\bigwedge_{(p,t_p,t_r)\in\left\{(p,t_p,t_r) \,\middle|\, \substack{B(p,t_p,t_r)\in\mathcal{V} \\ \wedge \\ t_r\notin\mathcal{U}\mathrm{R}(p)}\right\}} B(p,t_p,t_r) = 0 \tag{5.7}$$

Table 5.4: Summary of the supplementary rules related to the superfluous variables, in the context of a constraint problem with set of variables $\mathcal{V}$ and concerning the configuration validity w.r.t. universe $\mathcal{U}$.

$$\frac{}{N(t) \geq n \Rightarrow N(t) \geq n}\ \textsc{Component Type} \qquad \frac{}{N(p) \geq n \Rightarrow \left(\sum_{t \in \mathcal{UP}(p)} \mathcal{U}(t).\mathbf{P}(p) \times N(t) \right) \geq n}\ \textsc{Port}$$

$$\frac{}{\textbf{true} \Rightarrow \textbf{true}}\ \textsc{True} \qquad \frac{S_1 \Rightarrow C_1 \qquad S_2 \Rightarrow C_2}{S_1 \wedge S_2 \Rightarrow C_1 \wedge C_2}\ \textsc{Conjunction}$$

Table 5.5: The rules for translating of the configuration validity w.r.t. specification S into a constraint C ($S \Rightarrow C$).

Second: fulfilling the specification After encoding the final configuration validity rules in form of integer constraints, our next step is to encode in a similar fashion the input specification of our architecture synthesis problem. This will be much more straightforward than what we just did with the validity rules because our request language is already based on a collective approach to components and ports, and is not referring to particular components. Therefore the translation will be quite direct.

In order to easily convert the specification (which follows the syntax of our request language) into a constraint, we descend recursively into its structure, traversing the tree formed by the terms of type $S_1 \wedge S_2$ and create constraints corresponding to every leaf of form $N(t) \geq n$ or $N(p) \geq n$.

The recursive translation procedure that we follow for a given specification S when we want to obtain an equivalent constraint is very simple (for a formalized version see table 5.5):

- S = **true : no constraints.**

  The constraint corresponding to the specification S is **true**.

- $S = S_1 \wedge S_2$ **: a conjunction of two sub-specifications** S_1 **and** S_2**.**

  We recursively translate both sub-specifications S_1 and S_2, the constraint corresponding to S is the conjunction of the constraints corresponding to S_1 and S_2.

- $S = N(t) \geq n$ **: a specification leaf concerning a component type.**

  For a specification leaf of form $N(t) \geq n$, as the semantics of $N(t)$ in the specification matches exactly the semantics of the variable $N(t)$, the corresponding constraint looks exactly the same as this fragment of specification S (i.e. "$N(t) \geq n$").

- $S = N(p) \geq n$ **: a specification leaf concerning a port.**

  For a specification leaf of form $N(p) \geq n$, the corresponding constraint correlates the total arity of the port p provided throughout the configuration with the given constant n.

  The total provided arity of a port p is a sum of the arities of this port provided by all the components present in the configuration. We count it multiplying each of the $N(t)$ variables by the provide arity of the corresponding component type:

$$\left(\sum_{t \in \mathcal{UP}(p)} \mathcal{U}(t).\mathbf{P}(p) \times N(t) \right) \geq n$$

Full constraint problem

Now we have all the elements that we need to define formally the encoding of an architecture synthesis problem instance into a constraint problem:

Definition 22 (Encoding architecture synthesis problem into constraints)
Given an architecture synthesis problem with input universe $\mathcal{U}$ and input specification S, the corresponding constraint problem $\mathrm{P} = \langle \mathcal{V}, \mathcal{D}, C \rangle$ *is defined as follows:*

- *the set of variables*

$$\mathcal{V} = \bigcup \left\{ \begin{array}{lll} & & \bigcup\limits_{t \in \mathcal{U}_{dt}} N(t) \\ \bigcup\limits_{p \in \mathcal{U}_{dp}} & \bigcup\limits_{t_r \in \mathcal{UR}(p)} & \bigcup\limits_{t_p \in \mathcal{UR}(p)} \end{array} B(p, t_r, t_p) \right.$$

- *the function $\mathcal{D}$, attributing domains to variables:* $\forall v \in \mathcal{V},\ \mathcal{D}(v) = \mathbb{N}$
- *the constraints imposed on the variables $C = C_{\mathcal{U}} \wedge C_S$, where:*
  - *$C_{\mathcal{U}}$ is the constraint concerning the validity w.r.t. the universe $\mathcal{U}$, created using the translation provided in table 5.3,*
  - *and C_S is the constraint concerning the validity w.r.t. the specification S, created using the translation provided in table 5.5.*

Optimization

Let us now mention the final missing piece: optimization. Exactly as we have discussed in the context of the architecture synthesis problem (see subsection 5.2.6), instead of finding just any correct solution we often prefer to find the optimal one. In order to do that, we need to encode the optimization criteria from the architecture synthesis problem level (which consist of an *optimization function* and an *utility order*, as in definition 15 on page 77) as the optimization criteria on the constraint problem level (which is either $\max(e)$ or $\min(e)$ for a certain constraint expression e).

This is not feasible in general for any imaginable optimization criterion (as on the architecture synthesis level we can specify much more properties than on the constraint level[10]), but we can do that for the two concrete optimization criteria which we have already defined: *compact* and *conservative*. Before we have expressed them both as functions of the final configuration. Now we will translate them to a form of a constraint expression to minimize, using only the variables available in the generated constraint problem:

Compact The *compact* optimization function can be encoded very easily as a constraint expression. Before it was calculated by counting directly all the components present in the final configuration (see *compact* definitions in subsection 5.2.6):

$$compact(C) = |C_c|$$

Now we will count the number of components from a slightly different side. As we know, all the components present in the final configuration are of types defined in the universe, and the amount of components of each type corresponds to one of the $N(t)$ variables. Therefore in order to get the total number of components in the final configuration we can simply sum up all the values of the $N(t)$ variables in our constraint problem solution:

$$opt_{compact} = \min \left(\sum_{t \in \mathcal{U}_{dt}} N(t) \right)$$

[10] On the architecture synthesis problem level we can define our optimization criteria to correspond to an arbitrary ordering of the problem's solutions, while on the constraint level we are restrained by our constraint problem definitions (e.g. constraint syntax in table 5.1), which are less expressive than that.

Conservative The *conservative* optimization function is a little bit more problematic. Its aim is essentially to measure the distance between the initial and final configuration by counting the number of differences in components present in each of them (see *conservative* definitions in subsection 5.2.6):

$$conservative(C_{fin}) = \sum_{c \in C} \begin{cases} 1 & if \quad c \in dom(W_{init}) \wedge c \notin dom(W_{fin}) \\ 1 & if \quad c \notin dom(W_{init}) \wedge c \in dom(W_{fin}) \\ 1 & if \quad c \in dom(W_{init}) \wedge c \in dom(W_{fin}) \wedge W_{init}(c) \neq W_{fin}(c) \\ 0 & otherwise \end{cases}$$

This way of computing the number of differences, based on inspecting all the components one by one, is obviously not applicable to a constraint problem solution. It requires a notion of component identity, which simply does not exist in our constraint problem, where components of each type are treated collectively. We will thus replace it with a different counting mechanism, based on comparing the number of components of each type present in the initial and final configuration. We profit from the fact that:

- the number of components of a given type t in the initial configuration is known beforehand (and noted as $C_{init}(t)$),
- and the number of such components in the final configuration is equal to the value of the variable $N(t)$.

Thus our *conservative* optimization expression can compare all these values and sum up the differences:

$$opt_{conservative} = \min \left(\sum_{t \in \mathcal{U}_{dt}} |N(t) - |C_{init}(t)|| \right)$$

This collective method of counting differences by comparing the number instances of each component type is unfortunately slightly more restricted than the original one based on treating every component individually. They express the same thing only if we assume that:

- we always keep the existing instances of each component type rather than remove some of them and replace with new ones (i.e. the change in the value of $N(t)$ from a to b means exactly that: if $a > b$ then $a - b$ components of this type were removed; and if $a < b$ then $b - a$ new components of this type were created);
- no component changes type between the initial and final configuration and no removed component's name is used for a new component of a different type. (i.e. if there is a component with name c and type t in the initial configuration, then in the final configuration either it is still present and has the same type t or no component with name c is present at all).

5.3.4 Final configuration generation

Now we have arrived roughly at the middle of our constraint-based approach to solving the architecture synthesis problem.

In the preceding subsection we have covered the first step of the approach: converting an architecture synthesis problem instance to the form of a constraint problem P. Now we will assume, that the second step is also done: we imagine that we possess an external constraint solver, capable of finding a solution to the generated constraint problem, which has given us an (optimal) solution σ.

Therefore our current task is to complete the third step of the constraint-based approach by translating the constraint problem's solution back to the form of the original problem's solution, i.e. a final configuration represented in the Aeolus stateless model.

Stating the problem

Input First, let us state clearly again what information we possess at this point, which we can use in generating the final configuration. We can see all these in some sense as the inputs of the *configuration generation problem*:

1. The architecture synthesis problem level:
   - we know the inputs of the architecture problem instance, which consist of:
     - the input *universe* $\mathcal{U}$,
     - the input *initial configuration* C_{init},
     - the input *specification* S,
   - and, optionally, we also know the *optimization criteria* which should be used to select the optimal solution of this architecture synthesis problem instance.
2. The constraint problem level:
   - We also know the *constraint problem* instance $\mathrm{P} = \langle \mathcal{V}, \mathcal{D}, C \rangle$ corresponding to our architecture problem instance inputs (as in definition 22 on page 92)
   - and possibly also the *optimization criterion* $\min(e)$ or $\max(e)$.
3. Finally, we know the constraint problem solution σ, which is a correct solution of the problem P and, if constraint problem optimization criterion is defined, then this solution is the optimal one according to that criterion.

Output Our aim is to produce a final configuration $C_{fin} = \langle W_{fin}, B_{fin} \rangle$, which is a correct (as in definition 16 on page 78) and possibly optimal (as in definition 17 on page 78) solution of the given architecture synthesis problem instance.

Therefore, using the (optimal) constraint problem solution σ, we need to generate such a mapping of component names to component types W_{fin} and a set of bindings B_{fin} which will make the configuration C_{fin} a correct (and optimal) solution of the architecture synthesis problem instance.

What has to be done? The main issue that we are facing now is the fact that in our constraint problem components and bindings do not possess identity and are treated *en masse*, while in the final configuration they should be represented individually. All we have in the solution σ is the following information:

- variables $N(t)$ tell us how many components of each type t should be present in the final configuration,
- and variables $B(p, t_p, t_r)$ tell us what is the total number of bindings on each port p between all the providing components of type t_p and all the requiring components of type t_r.

Now we need to:

1. instantiate the components and give them identity (i.e. names), and
2. create correctly individual bindings between the instantiated components.

Preserving names What is also worth discussing is the question of the initial component identity, represented in the initial configuration by component names, which is lost during the translation to constraints (i.e. only when the initial configuration is not empty, as otherwise there is no information to be lost).

Although formally nothing obliges us to attempt restoring this identity, we should recall that in some sense the architecture synthesis problem is a sub-problem of the reconfigurability problem and thus we may see the final configuration as a transformation (by reconfiguration) of the initial one. Therefore some continuity between the two exists at the intuitive level. We will make an effort to reflect this continuity by preserving the component names between the initial and final configurations whenever possible. Additionally, doing this we will help us tremendously in implementing correctly the *conservative* optimization function.

Component Generation

Our first step to build the configuration $C_{fin} = \langle W_{fin}, B_{fin} \rangle$ will be to generate its component part W_{fin}.

Properties Let us recall that $W_{fin} : C \mapsto \mathcal{T}$ is a mapping form component names C to component type names $\mathcal{T}$. In order to make it correspond to the solution σ, we should simply make sure that for each component type $t \in \mathcal{U}_{dt}$ the mapping's image for exactly $\sigma\,(N(t))$ component names is equal to t :

$$\bigwedge_{t \in \mathcal{U}_{dt}} \sigma\,(N(t)) = \left| \left\{ c \mid W_{fin}(c) = t \right\} \right|$$

Also, when building W_{fin} we will try to *reuse* as many component names from the initial configuration C_{init} for each type t as possible in order to preserve the components continuity between the two configurations (and to comply with the *conservative* optimization function). Formally, this means that we will try to *maximize* the number of components that are present in both configurations and keep their type:

$$\sum_{c \in \left(dom\left(W_{fin}\right) \cap dom(W_{init})\right)} W_{fin}(c) = W_{init}(c)$$

Construction In order to respect these two properties, we construct the mapping W_{fin} in the following way:

- First we construct two sets, $\mathfrak{R}_t$ and $\mathfrak{G}_t$, for each component type t in the universe $\mathcal{U}$:
  - Set $\mathfrak{R}_t$ contains the names of components of type t which will be *reused* from the initial configuration C_{init}.

    It is the biggest subset of $C_{init}(t)$ whose cardinality is smaller or equal to $N(t)$. This means that if there are too many components of type t in the initial configuration then we remove some of them to get only $N(t)$ in the final configuration:

    $$\bigwedge_{t \in \mathcal{U}_{dt}} \begin{cases} \mathfrak{R}_t \subseteq C_{init}(t) \\ |\mathfrak{R}_t| = \min\left(N(t), |C_{init}(t)|\right) \end{cases}$$

  - Set $\mathfrak{G}_t$ contains fresh names for components of type t which will be *generated* in addition to the ones reused from the initial configuration C_{init}.

    Basically if there are less components of type t in the initial configuration than $N(t)$ then we keep all of them, and add new ones with the set $\mathfrak{G}_t$, making sure that we take *fresh* component names (so that they do not repeat):

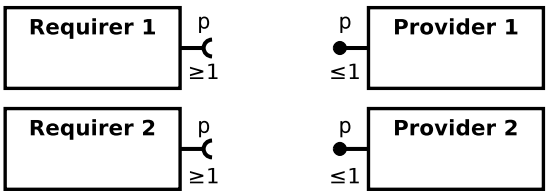
    $$\bigwedge_{t \in \mathcal{U}_{dt}} \begin{cases} \mathfrak{G}_t \subseteq C \\ |\mathfrak{G}_t \cup \mathfrak{R}_t| = N(t) \end{cases}$$

- In order to actually guarantee that all the newly generated component names are indeed fresh and unique, all the constructed sets $\mathfrak{G}_t$ should be *pairwise disjoint* and none of them should contain any *old* component names (i.e. present in the initial configuration):

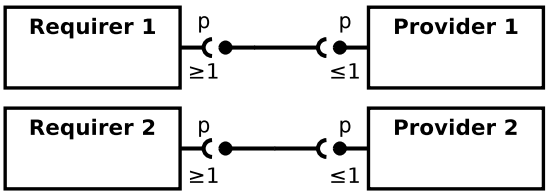
  $$\bigwedge_{t_1, t_2 \in \mathcal{U}_{dt}} t_1 \neq t_2 \rightarrow \left(\mathfrak{G}_{t_1} \cap \mathfrak{G}_{t_2} = \emptyset\right)$$

  $$C_{init_c} \cap \bigcup_{t \in \mathcal{U}_{dt}} \mathfrak{G}_t = \emptyset$$

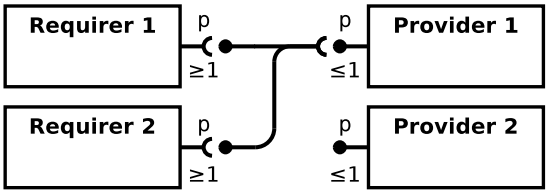
- Then using these sets the construction of W_{fin} is quite direct:



(a) The generated components.



(b) Correctly generated bindings: this configuration is valid.



(c) Incorrectly generated bindings: this configuration is invalid as the provide port of **Provider 1** is abused.

Figure 5.3: Example of a correct and an incorrect way to generate bindings which correspond to the same solution σ, such that $\sigma\left(B(p, t_r, t_p)\right) = 2$. We take: $t_r =$ "**Requirer**" and $t_p =$ "**Provider**".

– The components in C_{fin} are those in all the sets $\mathfrak{R}_t$ and $\mathfrak{G}_t$:

$$dom\left(W_{fin}\right) = \bigcup_{t \in \mathcal{U}_{dt}} (\mathfrak{R}_t \cup \mathfrak{G}_t)$$

– And all components in $\mathfrak{R}_t$ or $\mathfrak{G}_t$ are of type t:

$$\bigwedge_{t \in \mathcal{U}_{dt}} \bigwedge_{c \in (\mathfrak{R}_t \cup \mathfrak{G}_t)} W_{fin}(c) = t$$

The mapping W_{fin} constructed in this way satisfies all our requirements: it corresponds well to the solution σ and it maximizes the number of component names reused from the initial configuration. We will refer to this construction as our *component generation method*.

Binding Generation

Now, after generating the component part of the final configuration (i.e. the mapping W_{fin}), the time has come to create the bindings between those components: the set B_{fin}. This last step of the construction of the final configuration is fairly difficult.

Challenge The constraint problem solution σ only provides us with information about total number of bindings which should be created on a given port between whole groups of components of two given types.

This means that often a single solution corresponds to many different final configurations and in each of them bindings between these groups of components may be structured differently. Moreover, not all of these configurations have to be valid.

Let us consider a very simple example. We can imagine a situation presented on figure 5.3a, where we have:

- two components of type t_p providing port p with arity 1
- and two components of type t_r requiring port p with arity 1.

A solution σ such that $\sigma\left(B(p, t_r, t_p)\right) = 2$ is obviously right as it allows us to construct a valid final configuration, which we can see on figure 5.3b, where each component t_r is bound to a different component t_p. However, an invalid final configuration, presented on figure 5.3c, also corresponds to the same solution σ, while in the same time it violates the capacity constraints represented by the provide arity on t_p component bound twice.

As we can see, when generating the bindings for the final configuration we are guided in some sense by the constraint problem solution, but we still have to make sure that the bindings we construct are actually valid. Fortunately, even if it is not intuitively completely clear yet, every solution to a constraint problem constructed using our rules is guaranteed to always correspond to at least one valid final configuration (we will point to a formal proof of this property later on).

Binding variables Although the solution variables $B(p, t_r, t_p)$ are essential for assuring that creating correct bindings between the components in the final configuration will be feasible (i.e. that there is at least one valid configuration that we can construct which is valid), in fact nothing obliges us to directly use the values of these variables when generating the bindings. In other words, if we are able to find such a set of bindings between our generated components that gives us a **valid** final configuration, but one that does not match the values of the $B(p, t_r, t_p)$ variables, it is perfectly fine: our final configuration would be as correct as the one that matches these values.

Basically these variables have already served their primary purpose during the constraint solving phase and now we can ignore them if we think that we can do the job well using solely the information about the generated components and ports they require and provide (i.e. the mapping W_{fin} and the universe). In the algorithm which will be presented here we choose to neglect them, because in fact trying to respect their values only complicates the algorithm.

Thus basically the algorithm proposed here is based only on two inputs which are completely sufficient to generate correct bindings:

- information about the generated components and their types (available in the mapping W_{fin}),
- information about ports that each component type provides and requires as well as the corresponding require and provide arity (available in the universe $\mathcal{U}$).

Algorithm intuition Now let us present and explain the algorithm that we use to create bindings between the generated components. We call it the *matching algorithm*, as it matches the components which require bindings with those providing bindings in a way that respects the final configuration validity. Before introducing the algorithm itself, we need to state here one important observation:

- *Generation of bindings concerning ports p_1, p_2, etc. are always all completely orthogonal subproblems of the binding generation problem.*

  As bindings on different ports do not interfere with each other in any way, we can solve the problem of creating bindings for every given port p completely independently, without taking into account all the other bindings.

Let us now present the principle of the *matching algorithm*. We proceed, one by one, to create all the bindings concerning each port p in the following way:

1. We take (in any order) each component c present in the final configuration,
2. we check if it requires port p and with what arity n:
   - if it does not require port p then we go back to point 1 and take another component;
   - if it does, we continue:
3. we find a set of components providing port p large enough to satisfy the provide arity of c (i.e. we choose exactly n components providing port p, because that is enough),
4. and we construct all the bindings accordingly (i.e. one binding from the requiring component c to each of the providing components).
5. We repeat until all the components requiring port p have been treated.

The difficult part of this algorithm is the correct choice of the providing components at each time, such that in the end no providing component is bound too many times (i.e. the provide arity is not exceeded), and all the created bindings are unique (i.e. the binding unicity principle holds). If we can enforce these properties, then our algorithm, when completed for each port p, effectively assures that the generated final configuration is valid (i.e. all the requirements of components present in our final configuration are satisfied and no provide ports are overused). We achieve it by selecting the providers in the following way:

- always taking first the provider which has the most provide arity on port p left "unused" at the current point of the algorithm execution,
- and never taking the same provider twice for the same requiring component (in scope of each port p).

This way, by taking the providers in the descending order of "unused" provide arity without repeating, we effectively ensure that as many as possible *different* components proposing the same port stay available at each moment and thus can we always to match the remaining requirers with enough *different* providers.

Details In listing 5.6 we present the pseudo-code for our matching algorithm (formally proven correct in [31]). Let us explain now in detail what is the meaning of the used variables and how do the whole algorithm works:

- **Variable *provide_arity_left***

  Variable $provide\_arity\_left : \mathcal{U}_{dp} \rightarrow \wp\left(dom\left(W_{fin}\right) \times \mathbb{N}\right)$ is a function which initially assigns to each port $p \in \mathcal{U}_{dp}$ a set of pairs (c, n), such that for every component c which belongs to the final configuration and provides port p with provide arity n, pair (c, n) belongs to this set (and no other pairs belong there).

  When we are executing the algorithm, $provide\_arity\_left$ is used to keep track of all the yet "unused" provide arity of components providing a given port p, i.e. intuitively all the arity that is still available to be bound. With each binding created on port p with a providing component c the associated number n will be decremented by one.

- **Variable *generated_bindings***

  Variable $generated\_bindings : \mathcal{U}_{dp} \rightarrow \wp\left(\mathcal{U}_{dp} \times dom\left(W_{fin}\right) \times dom\left(W_{fin}\right)\right)$ is a function which assigns to each port $p \in \mathcal{U}_{dp}$ a set that contains all the bindings generated until a given moment by our algorithm. At the beginning the set corresponding to each port is thus empty and in the end the sum of sets for all ports contains all the bindings which will be present in our final configuration: $B_{fin} = \bigcup_{p \in \mathcal{U}_{dp}} generated\_bindings(p)$

```
1  // Selection of providers
2  select_providers(p, t_r) {                                          // selection algorithm
3    selected_providers ← ∅
4    for (c,n) ∈ provide_arity_left(p) in decreasing order of n {
5      selected_providers ← selected_providers ∪ {c}
6    } until |selected_providers| = U(t_r).R(p)
7    for c ∈ selected_providers { replace (c,n) with (c,n−1) in provide_arity_left(p) }
8    return selected_providers
9  }
10
11 // Bindings generation for a given port p
12 generate_bindings(p) {
13   provide_arity_left(p) ← {(c,n) | W_fin(c) ∈ UP(p) ∧ n = U(W_fin(c)).P(p)}   // initialization
14   generated_bindings(p) ← ∅
15   for c_req ∈ W_fin {                                                 // main algorithm
16     if W_fin(c_req) ∈ UR(p) {                                       // invariant position
17       selected_providers ← select_providers(p, W_fin(c_req))
18       generated_bindings(p) ← generated_bindings(p) ∪ {(p, c_req, c_prov) | c_prov ∈ selected_providers}
19     }
20   }
21   return generated_bindings(p)
22 }
23
24 // The main function
25 main() {
26   B_fin ← ∅
27   for p ∈ U_dp {
28     B_fin ← B_fin ∪ generate_bindings(p)
29   }
30   return B_fin
31 }
```

Table 5.6: Binding generation algorithm.

- **Variables invariant**

  The following invariant is always true when we pass through the *invariant position*: for every port p, if we subtract the already generated bindings on port p present in the $generated\_bindings(p)$ set from the provide arity of port p provided in the beginning by each component c, we will get the current "unused" provide arity of port p for the component c, equal to the n in the corresponding (c, n) pair present in $provide\_arity\_left(p)$.

- Selecting providers subroutine: **`select_providers(`**p**`,`** t_r**`)`**

  We have a component of type t_r which requires port p (we call this component a *requirer*) and we want to create bindings which would satisfy this requirement. In order to achieve that we need to select a number of components that provide port p (we call these components *providers*) equal to the require arity of the component type t_r. We do it by filling the variable $selected\_providers$ with the set of providers:

  1. We start with an empty set (in the line 6). Then we take elements from the set of pairs $provide\_arity\_left(p)$, starting by the greatest "unused" provide arity n and descending (line 7), and we add the component c from each pair to the set $selected\_providers$ (line 8). We do it until we have enough providers: a number equal to the require arity which we want to satisfy (line 9).
  2. Then (in line 10) we adjust the set $provide\_arity\_left(p)$ by subtracting one from the n of each pair (c, n) corresponding to a providing component that we have selected (i.e. added to the $selected\_providers$).
  3. Finally we return the set of selected providers accumulated in the variable $selected\_providers$ (in line 11).

- Generating bindings subroutine: **`generate_bindings(`**p**`)`**

  We want to generate the bindings on port p between all the components which require and provide this port, so that the requires are satisfied and provides are not overused. We achieve it by taking one by one every component present in the final configuration which requires port p and creating bindings which satisfy its requirements:

  1. First we prepare the initial value of $provide\_arity\_left(p)$ (in line 13) which contains information about the provide arity of all the components providing port p.
  2. We start with an empty set of bindings (in the line 14). Then we take each the component c_{req} present in our final configuration (line 15) and if it requires port p (checked in line 16), we generate bindings satisfying its requirements:
     (a) we find a set of components providing port p large enough to satisfy the provide arity of c using our selecting providers subroutine `select_providers(`p`,` t_r`)`,
     (b) then we construct the new bindings (line 18): one binding from the requiring component c to each of the providing components
     (c) and we add them to the already generated bindings on port p (still line 18).
  3. Finally we return the generated set of bindings concerning port p (in line 21) so that it may be merged with bindings on the other ports.

- **Parallelization**

  For each port p present in the input universe $\mathcal{U}$ the binding generation can be performed independently, thus the for loop in the main function (in line 27) could be parallelized.

Conclusion

Now we can put the whole configuration generation in one definition and state precisely what we can call a final configuration generated using the constraint solving approach which corresponds to a given architecture problem instance and a given constraint problem solution:

Definition 23 (Generating the final configuration for the architecture synthesis problem)
In the context of a certain architecture synthesis problem instance and given a variable mapping σ, which is a correct solution of the constraint problem P *corresponding to this architecture problem instance (as in definition 22 on page 92), we define each configuration $C_{fin} = \langle W_{fin}, B_{fin} \rangle$, such that:*

- *the mapping W_{fin} was build using our* component generation method
- *and the set B_{fin} was build using our* matching algorithm*,*

as a final configuration generated using the constraint solving approach that corresponds to this architecture problem instance and this solution σ.

5.3.5 Conclusion

Let us now put it all together and define more formally how does the constraint solving approach work.

Definition 24 (Solving architecture synthesis problem using the constraint solving approach)
In order to solve an architecture synthesis problem using the constraint solving approach, given an architecture synthesis problem instance defined by following inputs:

- *an initial configuration C_{init},*
- *a universe $\mathcal{U}$,*
- *a specification S,*

first we create the constraint problem *instance* P $= \langle \mathcal{V}, \mathcal{D}, C \rangle$ *corresponding to the architecture problem instance inputs (as in definition 22 on page 92),*

then we feed this problem instance to an external constraint solver and we obtain a variable mapping σ, which is a correct solution of the problem instance P *(as in definition 19 on page 82)*

finally we generate the final configuration C_{fin} corresponding to this architecture problem instance and the solution σ (as in definition 23 on page 101).

Definition 25 (Finding an optimal solution of architecture synthesis problem using the constraint solving approach)
In order to find an optimal solution of an architecture synthesis problem using the constraint solving approach, given the inputs of the architecture synthesis problem instance and given:

- *formalized* optimization criteria *(see definition 15 on page 77) which induce a total preorder on configurations $\left(\Theta, \leq_{opt}\right)$,*
- *a constraint problem* optimization criterion *opt inducing a equivalent preorder $\leq_{opt}$ on solutions, i.e. such that for every two correct solutions σ_1 and σ_2 of the problem* P *and their corresponding final configurations C_{fin_1} and C_{fin_2} (generated as in definition 23 on page 101) the relation between these solutions is the same as the relation between these configurations in the respective preorders:*

$$\sigma_1 \leq_{opt} \sigma_2 \iff C_{fin_1} \leq_{opt} C_{fin_2}$$

We apply the method described in definition 24 on page 101 with one difference: when we feed the constraint problem instance P *to the constraint solver we include also the optimization criterion opt, so that the solution σ which it generates is optimal according to this constraint criterion (as in definition 21 on page 83).*

Our final configuration C_{fin} enjoys several properties: soundness, completeness and optimality (proven in [31]):

- **Soundness**

  The found configuration C_{fin} validates the input universe $\mathcal{U}$ and input specification S.

- **Completeness**

  If there exists a configuration C_{fin} that validates the input universe $\mathcal{U}$ and satisfies the input specification S then using this approach we will successfully compute some final configuration C'_{fin}.

- **Optimality**

  If some optimization criteria were provided, the generated configuration C_{fin} is optimal according to the chosen optimization criteria.

5.4 Conclusion

In this chapter we have developed a stateless variant of the Aeolus component model and used it to define and describe the architecture synthesis problem. Then we have presented and explained in detail our approach, based on constraint solving, which lets us find the optimal solutions for instances of this problem. All the elements of our model and approach have been formalized and proven correct.

What was introduced here will serve as a base for the following chapters, where we will extend and adapt both the model and the approach in order to, finally, apply all that in practice by building a real working tool around the established theory.

CHAPTER 6

SYNTHESIS OF A CLOUD-BASED ARCHITECTURE

6.1 Motivation

After presenting the Aeolus model for distributed systems, with its main theoretical properties and limiting results, in chapter 4, we have introduced a reduced, stateless version in chapter 5. For this stateless version of the model, we have shown that the *architecture synthesis problem* can be solved by using an approach based on constraint programming.

In this chapter, we build on this positive result and extend it in order to make it applicable in practice, taking into account many extra details that occur in a real distributed environment, like a computer network or a cloud, and that were up to now abstracted away in our models. For this, we will extend the stateless Aeolus model and readjust our approach as needed.

My contribution

A detailed version of the work presented in this chapter can be found in the technical report *Optimal Provisioning in the Cloud* [31] and a short summary has been published in the article *Automated Synthesis and Deployment of Cloud Applications* [24].

I participated in the conceptual part of the work, together with the Aeolus project team in Paris: Michael Lienhardt, Roberto Di Cosmo, Ralf Treinen and Stefano Zacchiroli. Moreover, I am the main author of the Zephyrus tool that implements these theoretical ideas, and several important practical optimizations (discussed further in chapter 7).

6.2 Introduction

6.2.1 What we have

Until this point we have developed quite a lot of theory related to modelling distributed systems and problems concerning their administration.

Formal model

We have conceived a strict formal model of a distributed system, the Aeolus model, in two basic variants:

- The full stateful variant of the Aeolus model (presented in the chapter 4).

  It permits us to model the distributed system and its reconfigurations, it is reasonably expressive and yet quite simple.

- The stateless variant of the Aeolus model (presented in the chapter 5).

  On the one hand it is less expressive than the full variant, as the whole dynamic aspect was left out of it and there is no notion of system reconfigurations.

  On the other hand, it is well adapted to the situations where we are only interested in the static aspect, like when solving the architecture synthesis problem, where the dynamic aspect of distributed systems is not considered on purpose.

Results

We have reached some interesting theoretical results regarding these two model variants:

- mainly limiting theoretical results concerning the stateful model and especially the related *reconfigurability problem* and *achievability problem*,
- but also some encouraging results for the stateless model: we have developed a method of solving the *architecture synthesis problem* using an approach based on constraint solving.

6.2.2 What do we want now?

Now we would like to get our theory closer to practical applications in order to be able to use our model and approach in a real distributed environment. We will consider two broad types of distributed environments: a **computer network** and a **cloud**.

By **computer network** we simply mean here a (fixed) number of interconnected physical machines, which all belong to us and which may potentially be heterogeneous (i.e. all can possess different amounts of computing resources, run different operating system, etc). By **cloud** we mean something similar to what is called IAAS[1], either in a public cloud (e.g. Amazon EC2) or a private one (e.g. implemented on OpenStack).

These environments can be intermixed in practice, in what is usually called a *hybrid cloud* where part of the distributed application is placed on the local network, and part is deployed in an external cloud.

Our aim is thus to use our architecture synthesis approach based on constraint solving in these kinds of environments. First we will adapt the theory to work well in this context and then we will put that theory into practice.

"Cloud" disambiguation

As the word "cloud" and the expression "cloud computing" can be nowadays interpreted in a plethora of different ways, let us state what it means for us here exactly. Basically, we use the *cloud computing* concept solely at the level of virtual hardware. A *cloud* definition for us, in the context of this document, would be: *a computer network where we can dynamically add and remove machines*. We will assume two things about the machines available in our clouds:

- they may be heterogeneous, but each of them realizes one of certain *predefined machine types*, whose parameters (e.g. amount of computing resources, selection of available operating systems, etc.) are known beforehand[2];
- each machine has a certain attributed *cost*:

[1] Infrastructure As A Service [11]

[2] This corresponds to the situation, where we have a given limited selection of available virtual machine images that we can use, be it in a public or a private cloud.

– in public clouds this cost is well defined by "money per hour" (although sometimes the calculation may be more complex, e.g. if we can choose among different subscription plans);
– in private clouds we need to define this cost ourselves, depending on how we estimate the value of our computing resources.

6.2.3 Our approach

Let us assess now the "what we already have" in the context of "what we want". The question which we are posing here is: how can our model and approach be used in order to attain our new aim?

The component placement problem

The main issue which we will have to deal with is the *component placement problem*. It stems from the fact that in the real world software components are installed on actual machines, whereas in the Aeolus model (both in the stateful and the stateless variant) we do not make a distinction between different machines at all. Our model is completely flat and puts all the components on the same level, in one big "distributed system" bag with no notion of separate locations.

We avoided confronting this problem until this point and we did that on purpose. In fact one of our initial assumptions when designing the Aeolus model was to abstract the machines completely from the formal model and focus on the software components and their relationships. Consequently, we know which components are in the system and how they work together, but we do not know how they are distributed on the underlying architecture (be it real or virtual machines).

Addressing the component placement problem

Now the time has come to address the component placement problem. We need to integrate in into our approach in order to make the approach useful in the new context. On the basic level, there are two alternative ways of attempting to do that:

1. **Sequential composition**

   We could simply decide to place the components on machines **after** synthesising the architecture. The idea is to first use directly the existing method to synthesise a flat system configuration (i.e. all the components with the bindings between them), and then determine how to place the components on the machines.

2. **Direct integration**

   Or, otherwise, we could include the machines in our model and make the component placement an integral part of the architecture synthesis.

The **sequential composition** of these two problems seems like a much more advantageous option and it would be our first choice if only we could take it. Unfortunately, after closer examination, we are forced to abandon this route and stick with the **direct integration**. Let us explain why.

Why does sequential composition not work very well? In principle the first sequential composition idea could work well if both our problems (i.e. architecture synthesis and component placement) were orthogonal to each other. Then we could compose them sequentially without any trouble. Unfortunately it is quite easy to see that, the optimal solution of the architecture synthesis problem often depends on the given machine park. Thus there is a connection between the two problems and so they are not orthogonal.

Intuition The reasons behind this fact are intuitively rather obvious:

- In a fixed **computer network** or a **private cloud** we have limited computing resources. As the components deployed in the system are using computing resources, the availability of the resources may influence the choice of components which we can install.

- In a public **cloud** on the other hand we can imagine having virtually infinite computing resources at our disposal. The issue persists though, because these resources are associated with virtual machines, and using these machines has a cost, which we usually try to minimize. Therefore the decisions about what components we put in the system may influence the total cost of deploying this system. Thus in order to minimize this cost we need to take into account the resource consumption of our components (which is related to the component placement).

Hence in both cases intuitively the optimal solution of the architecture synthesis problem depends on the placement of components on the machines.

Example Let us give a more concrete example showing a situation where sequential composition of these two problems will not work well. We will consider a huge computation-heavy service and our task is to deploy it successfully in a given distributed system. We imagine that we can deploy it in two different ways:

- either in **one piece**, which corresponds to a single "huge" (figuratively, i.e. consuming a huge amount of resources) component,
- or in **multiple pieces** as a distributed master-slave system, which corresponds to many "smaller" components.

Of course deploying it in one piece requires a machine which is "huge" enough for its needs (i.e. provides enough resources), so it is only possible if we actually have such a machine at our disposal.

Now let us consider separately two cases of distributed environments, a fixed computer network and a cloud:

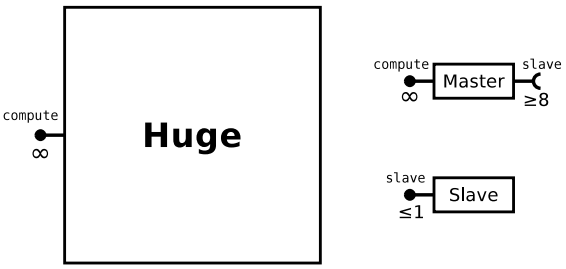
- in the **fixed computer network** case we may simply not have a machine "big" enough in our computer network to deploy the single huge component (see figure 6.1c),
- in the **cloud case**, deploying the distributed version on many "standard" machines (see figure 6.1b) may be much cheaper than deploying the single-component version on a single "huge" machine (see figure 6.1d).

Therefore in both cases we can come up with such a machine park example, that a solution of the respective architecture synthesis problems which does not take into account the restrictions coming from the component placement issue may be either completely impossible to deploy or deployable but suboptimal.

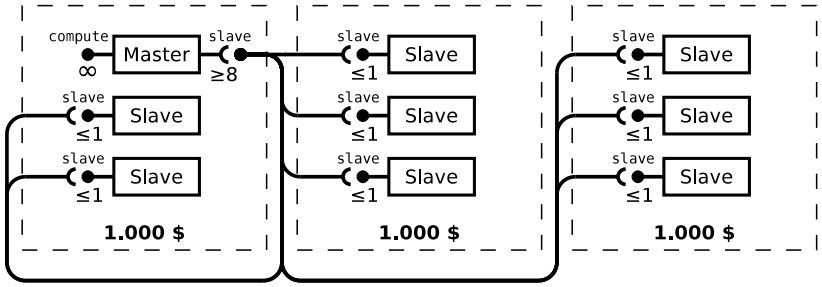
Conclusion So, as we can see, this approach would be at best very difficult to pull off. If we really wanted to put this idea into practice (i.e. attempt to find a solution to the architecture synthesis problem without taking the machines into account, and only after that try to place the components on the machines) we should anticipate that in some cases this may fail:

- Sometimes we will simply not be able to place the components that we have generated on the available machines (like in the example presented just before with a "huge" database component and a machined park of little machines).
- And sometimes the component placement that we can do will be suboptimal, which means that although we could find a way to place the components on the machines correctly, there would exist another solution of the given architecture synthesis problem instance, which could let us place the components in a way that results in a better use of resources and a less costly deployment (like in the example before with a cloud).

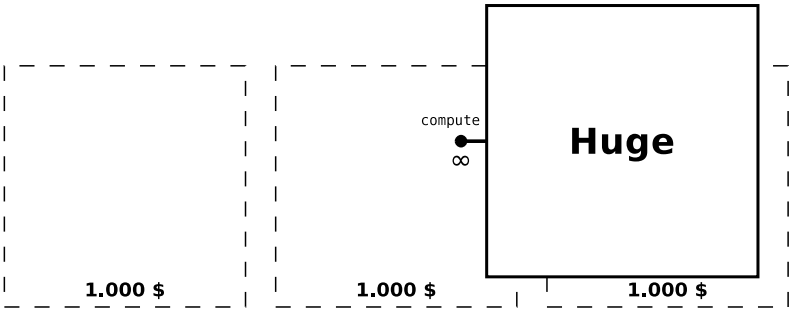
The only reasonable thing to do in both these cases would be to somehow introduce a feedback loop going back from the component placement problem phase to the architecture synthesis problem phase (e.g. "this solution may be correct, but is not deployable on our machine park, try again!" or simply "try finding



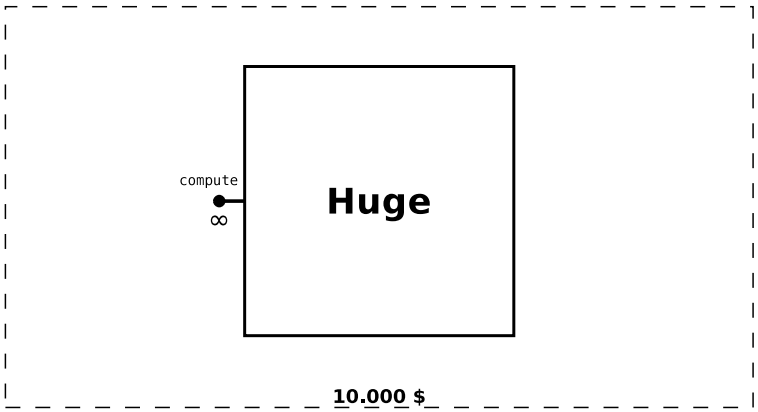
(a) Our universe: the unified version (component type **Huge**) and the distributed master-slave version (component types **Master** and **Slave**), both provide the same thing.



(b) Network case / cloud case: everything fits on 3 standard machines using the master-slave version (components of types **Master** and **Slave**).



(c) Network case: the unified version (component of type **Huge**) cannot fit on a standard machine.



(d) Cloud case: the unified version (component of type **Huge**) fits on a huge machine, but this deployment is much more expensive than the one in figure 6.1b.

Figure 6.1: Example of a situation where component placement affects architecture synthesis. We can deploy our main service either as a single unified component "**Huge**" or in a distributed master-slave version with components "**Master**" and "**Slave**)"

a solution closer to the optimum if you can!"). But then we would need to guide the solver by some means during the architecture synthesis problem phase in the direction of a deployable solution (in case if the current one is not deployable); or, which seems even more difficult, we would need to guide it in the direction of a better solution in order to converge eventually to the optimal one (i.e. the one that is closest to the optimum among all the solutions which are deployable).

We choose direct integration Instead we decide to attempt to include the machines and the component placement problem directly in the architecture synthesis problem. In order to do this we will have to change the model and adapt all the parts of our approach which dependent on it, like the constraints generation, accordingly.

Unfortunately there is at least one downfall of this idea which we can easily anticipate beforehand: making the model more complicated will surely increase the complexity (in the sense of difficulty to solve for the constraint solver) of the architecture synthesis problem. However, this still seems to be our best bet, therefore we choose to follow this track and see how it works out.

6.2.4 Conclusion

Let us summarize the decisions which we have taken so far and that we will attempt to realize in the rest of this chapter. Basically in order to attain our new objectives, we choose to keep our approach and adapt it accordingly.

We have decided to stick to the principles of our approach, as it was described before in chapter 5, and modify it by integrating new elements into it (i.e. incorporating the component placement problem), instead of, for example, using it as a part of a different approach (e.g. placing the components only after synthesising the architecture). Therefore we will have to extend our model and refine all the parts of the approach which depend on it (mostly the constraint problem generation) accordingly.

6.3 Extending the architecture synthesis problem

At the end of the previous section we have decided that the best way to proceed right now is to incorporate the *component placement problem* into the *architecture synthesis problem* and then adapt our constraint-based approach to work well with this new *extended architecture synthesis problem*. In order to realize this plan we must first extend the model (i.e. Aeolus stateless model) which we use in the approach, at the same time taking into account how will our extensions work with the rest of the approach (especially the constraint generation part). Basically the structure of the architecture synthesis problem will stay the same, while the model behind it will change[3] and our approach will be adapted accordingly.

The main theme of this section is: what elements do we need to add to the model exactly? Our aim is to introduce the ability of placing components on machines to our approach. The components are already present there, we will thus start by introducing the concept of machines to our model and then link machines with components. As before, we will try to make the model expressive enough for our purpose and at the same time keep it as simple as possible.

6.3.1 Machine park

We will call the machines represented on the level of our formal model *locations*. This term is quite generic and we define it as "a place where components can be installed", thus it easily covers both physical and virtual machines.

We could say that before we had a *flat model* with a 0-level location hierarchy: all the components were residing in the same single flat space. Now we want to introduce a basic 1-level hierarchy: we will have many different separate locations to place our components in.

In our new model we should be able to define the *machine park*, describing the machines in our distributed system:

[3] For example, there will still be an universe in the inputs, but in the previous version of the architecture synthesis problem it was a *stateless Aeolus model universe* and in the extended version it will be an *extended stateless Aeolus model* universe.

- it may be a static one, representing a computer network of physical machines;
- it may be a dynamic one, representing a (public or private) cloud of virtual machines;
- or possibly it could be a mixed one (e.g. a few physical machines plus a private cloud).

Every component present in a configuration will be allocated to a single location, which represents it being installed on the corresponding machine.

6.3.2 Component co-location

We cannot stop at adding the machine park representation to our model and establishing a link between the locations and the components (which we could call a component placement relation). If we do only that, then there would be no reason at all (at the model level) to not put all the components on a single machine.

In practice there are different points of view and various factors that we take into account when deciding about the component placement in distributed systems:

- we consider the components individually (i.e. each separately):
  - *Which component can / should go where?*
- and we consider the components collectively (i.e. in the context of their relations with each other):
  - *Which components should be together?*
  - *Which components should not be together?*

In other words we should consider both what we *can* put (together) on a given machine (i.e. what is possible to install where and what is possible to co-locate where) and then we should consider what we *should* put together (i.e. what components we prefer to be co-located).

What components can we put together?

Individually Not all components are installable on every available location:

- First, some software may be only compatible with certain operating systems or only installable on particular versions of certain operating systems (e.g. a selection of Linux distributions in appropriate versions).
- Also, as each machine's capacity in terms of computing resources (e.g. CPU power, RAM) is limited, not all components may be able to run on all machines: their hardware requirements may be higher than what the machine provides.

Collectively The question of coinstallability of (two or more) components is more complicated:

- Some components may obviously not be coinstallable on any machine because they are compatible with different operating systems.
- But it may also happen, that although both components are installable separately on a certain operating system, when we try to install them together (on this system) we will find that it is impossible (see [80, 25]). This may occur for example when they depend on conflicting software packages (e.g. two mail servers, like Sendmail [107] and Exim [58]; or two versions of the same service, based on different versions of the same software package[4]).

[4]Many software distributions do not allow multiple versions of the same package to be installed together.

- Sometimes, components may get coinstalled without trouble, but then do not work correctly in practice for the reason of being co-located, like when we their collective hardware requirements are higher than what the machine provides (e.g. we have put several computation-heavy services on a small machine that cannot handle so much computing).

These can be all classified as natural causes for component incompatibility, induced by actual obstacles which make installing certain components together impossible (either always impossible or impossible only on particular machines or operating systems). We should add to this list all the possible artificial restrictions imposed by the system administrator, which we will simply call *deployment policies*. The system architect or administrator may have some more sophisticated reasons to not want certain components to be co-located together, like for example redundancy and fail safety concerns (i.e. minimizing the consequences in case a machine fails).

What components we should put together?

Now let us consider the opposite question: what components should be installed together on a single location. Although there may exist some local dependencies between two components, where one of them requires the other to be installed on the same machine, this kind of dependencies is rather discouraged in the Aeolus philosophy, so we are very reluctant to introduce them directly into the model.

However, more generally it often happens in practice (i.e. when distributed systems are designed manually) that there exist some kind of *affinity* between certain components. For some reasons we are more satisfied when particular components are installed together (e.g. when two components communicate a lot).

6.3.3 Decisions

We have listed here quite a lot of different features related to the question of component placement and co-location. Some of these are essential if we want to address the component placement problem in a reasonable way, while others are interesting but optional. The difficulty of including these features in the Aeolus model and in the whole approach (as we should take into account that we will have to encode them into constraints at some point) also varies. Now we need to prioritize and decide which of them we will include in the model and to what extent.

Machine park

It is crucial for us to model the available machines and at least their essential properties: operating system, computing resources and cost.

Component co-location

Local conflicts The priority for us will certainly be modelling the hard restrictions on component co-location: what cannot or should not be put together on a single machine. That is simply because we want to be sure to synthesise a correct configuration.

Local dependencies On the other hand modelling concepts like component affinity is much less important, due to various reasons:

- Local dependencies between components are discouraged in our model for a reason, it was not designed and it is simply not well-suited for strong "what should be together" constraints.
- Also, there are some deeper issues in our approach which would make it quite difficult to implement this kind feature if we decided to do it, like for example the issue of local bindings.

  Let us recall that on the constraint problem level we do not have the information about the identity of each component, as we treat all components of a single type collectively. In practice it means that, although we could be potentially able to guarantee that certain types of components would be put together, we have no way to assure that they would be bound with each other locally and not with some other components on different locations.

- That said, the concept of affinity between components, if understood not as a hard dependency, but only a preference, seems interesting and might in fact work. Instead of hard constraints forcing components to be together, we could introduce an affinity factor between components which could then be optimized for. Although it sounds interesting, trying to implement that is certainly not our highest priority (and the local binding problem remains), so we will not pursue this idea any further here.

6.4 Extending the stateless model

In this section we will explain in detail how we extended the stateless Aeolus model in order to adapt it to the new context: the extended architecture synthesis problem, which includes now the issue of placing components on machines. Design decisions presented here follow directly the considerations discussed in the previous section.

In fact everything that was present in our approach before (as it was described in the chapter 5) will remain there. This includes both the formal model itself as well as the constraint-related part of the approach. In other words, we will not remove anything, we will only add new features, both at the model level (i.e. we will strictly enlarge the model) and at the constraint solving level (i.e. we will add new variables and new constraints).

In order to explain these modifications, first we will start by providing an informal intuitive description and a semi-formal definition of every new feature that will be inserted into the model. Then, after presenting all the new features, we will provide a full formal definition of the extended model at the very end of this section.

6.4.1 Locations

First thing that obviously needs to be added to our model is a way to represent the machines in the distributed system. Thus we will begin by introducing the concept of *locations* to our model, which will correspond directly to the available machines. Now our distributed system *configuration* will specify not only the *components* present in the system and the *bindings* existing between them, but it will also include the *locations* where the components are placed.

Definition

Let us discuss in detail what we exactly mean when we say that the locations represent the "available machines" in the context of our extended model:

- Locations can represent physical or virtual machines. What we mean here by a **machine** is basically "anything where we can install some components of our distributed system".
- Intuitively the fact that a machine is **available** to us means, that we have some kind of sufficient administrator rights on it, so that we can actually install some components of our distributed system there. The exact amount of control which we can exercise over each machine is not strictly defined:
  - we can imagine that on one extreme of control we would put machines which we control more or less totally (e.g. we can change their operating system if we choose to do so);
  - and on the other extreme we would put the machines where we do not even have a root account, but we can still install certain components, e.g. a big mainframe computer which we can access for some purposes, but is not really under our control.

Implicit assumptions

We also assume implicitly that all these machines fulfil certain additional conditions, which are necessary in order to be sure that what happens on the model level (i.e. the system reconfigurations) can be actually performed on the system level (i.e. the corresponding actions have sense and can be really performed):

- All the machines should be easily **accessible** for the administration tasks (e.g. we can login through SSH), so the system administrator can actually perform the actions corresponding to installing or removing components, changing their internal state, etc.
- All the machines are **interconnected** in a way that our components on different locations can bind with each other freely.

  This usually means that the machines represented by our locations are in a computer network. They may all be in the same local network or they may all be simply connected to internet: it does not really matter which kind of connection is it as long as the condition of free communication is satisfied enough to let them form bindings. For example, if two machines are both connected to internet, but are in different IP address spaces and thus have to interact through NAT[5] they might not be able to communicate freely enough to establish bindings.

Heterogenity

The machines represented by our locations can be heterogeneous: they may provide a different amount of computing resources, run different operating systems, and also we can a have different level of control over them, etc. We do not aim to represent all these aspects of the heterogeneity of our machines on the model level. What is important for us is strictly the question of how each machine works with our different component types.

For example, difference in the amount of RAM is often very important in this matter, so we will usually include this parameter in the model. The graphic card's memory probably is much less relevant, thus in most cases we will threat machines with great graphic cards exactly the same as the ones without graphic cards. But if, for instance, some of our components perform heavy cryptographic computations using the machine's GPU[6] (e.g. through CUDA [92]), then this difference may suddenly become important and worth modelling.

Components on locations

Let us now see how do we model the component placement by linking the components with locations. Each component present in a *configuration* must be allocated to a single *location*. This corresponds to all the software and data which form this component being installed on the machine represented by the given location.

Installability and coinstallability Not every type of component needs to be installable on every location, because of various reasons mentioned before, like the operating system, computing resources, etc. Also, the way that the components are co-located on machines is far from unconstrained: there are many factors which may affect this question. We will explain shortly how do we model this aspect of the component placement problem (i.e. what can be (co)installed where) in detail.

One location per component We have decided exactly, that each component *must* be attributed to a *single* location. Why do we not allow for components which are not attributed to any location? Or for components which are installed on many locations? Well, both these ideas simply do not correspond to what a component fundamentally represents for us:

- First, components should never span several machines (see chapter 4) because of our basic component unity principle.
- Second, components are supposed to correspond to some actual pieces of running software. And software needs some hardware to run on.

  Note: We could imagine, that it might be potentially useful in some circumstances to allow in our model for something like *virtual components*, which do not correspond directly to any actual piece of software and thus

[5] Network Address Translation [110]
[6] Graphics Processing Unit

are not attributed to any location in particular. But this idea does not fit well the principal way of understanding the components that we have established so far, so we will not apply it.

Fixed machine park

We have mentioned in the previous section that we would like to be able to model not only static, but also dynamic machine parks (i.e. clouds) in our approach. The dynamic aspect here means that we can add or remove some machines from the given machine park at will.

Unfortunately, implementing this feature directly on the model level would cause some serious trouble further down the road, as dynamic machine parks would interfere with our constraint generation method. On the constraint problem level we need to know exactly how many machines are there and we need to be able to identify them in order to properly handle the component installability and coninstallability issues.

Therefore, instead of trying to handle directly machine parks with variable number of machines, we will restrict ourselves to fixed machine parks and try to simulate the dynamic aspect. We will always fix the available locations in each instance of the problem: no locations can be added to the ones defined in the initial configuration nor removed from them.

In order to represent a cloud, we will model a lot of different locations corresponding to virtual machines that *might* be deployed in this cloud if needed. Then in the solution (i.e. the final configuration) some of these machines will be used and others left empty. In practice we will consider the used ones (with at least one component installed on them) as "machines that must be provisioned" and the empty ones as "machines that should not to be provisioned" which thus can be simply ignored.

This solution surely puts some additional weight on the constraint solver (because of a potentially big number of locations to choose from), but is feasible and quite straightforward. We do not know if a viable solution based on including the dynamic machine parks directly in the model exists at all.

Semi-formal definition

Let us define the location concept semi-formally. In order to model the locations and placement of the components on locations, we extend our stateless formal model model in the following way:

- Every *configuration* now contains a set of *locations*.
- Each *location* has:
  - an unique *name*,
  - some additional individual properties (to be defined).
- **Every** component present in the configuration must be assigned to **exactly one** location.

In the context of the extended architecture synthesis problem, the sets of locations are **fixed** between the initial configuration and the final configuration. This means that there must be exactly the same set of locations in the initial and final configuration of every correctly solved problem instance. This will be represented as a new final configuration validity condition.

The question of keeping the individual properties of each location (i.e. to what degree each location is required to stay unchanged) in this context stays open for now. For now we assume that at least the names of the locations must remain the same in both sets, because they constitute fundamentally the locations' identity. Probably some of the other properties will be fixed and some mutable, we will discuss it when defining these properties.

6.4.2 Repositories and packages

Next two elements that we will add to our model are *software repositories* and *software packages* [35]. These two concepts are supposed to help us in addressing one aspect of the component placement problem (installability and coinstallability of components on machines).

Repositories

In the extended model we will attribute a *repository* to each location. This repository corresponds more or less to an operating system installation on a given machine.

Restricting to package-based systems In practice, we will restrict our attention to machines on which package-based operating systems are installed, typically GNU/Linux distributions, like Debian [29], Ubuntu [20], Mandriva, Suse, etc. as well as many other Unix based systems.

Concentrating on package-based systems greatly helps us in determining the relationships between different services installed on a single machine, providing we can map our services to software packages which they depend on. Our focus on these particular systems is justified by the fact that many modern distributed systems are built using FOSS[7] on various versions of GNU/Linux (like all the popular LAMP[8] web application deployments).

All the developments described in this chapter are also adaptable to non-package based systems, but one would need to build from scratch a detailed description of the software components available, as well as their relationships to the services running on a machine.

Components and repositories As we have hinted many times before, there is not only a relation between repositories and locations (i.e. each location has a repository) and a relation between locations and components (i.e. each component is placed on a location) , but also a relation between repositories and components: as not every service can be installed and run on every operating system, each component type can be installable or not on a given repository.

This relation will not be defined in our model directly. There will be one more layer present there between repositories and components: this layer will correspond to software packages and we will get to it shortly.

Should locations have fixed repositories? In the meantime, there is yet another important question to be asked here: on the model level, should each of our locations have a fixed repository or should it be possible to change it?

As we have established, the locations present in the initial and final configuration should always be the same. However, we left the decision about exactly which of their properties were supposed stay the same for the moment of defining these properties. Location's repository is considered as one of the location's properties. Should locations always keep their initial repository or can it be changed to a different one in the final configuration?

For now we simply decide, that the repository can be changed freely. However, we acknowledge that this may make sense in some cases and no sense in others. Basically it depends on the question of how much control do we have over the machines. If we have a total control, then we can change their operating system, otherwise it is not so sure:

- thus in case of a private cloud it makes a lot of sense, because we will be usually able to choose the operating system of our virtual machines freely at least among several available alternatives (in a public cloud more or less too);
- in case of physical machines on the other hand, often the restrictions coming from the particular local context will decide that we cannot change the machine's operating system.

For now we decide that we want to prioritize modelling cloud environments and thus we will always permit changing the location's repository between the initial and final configuration. However, we imagine it would be easy to fix them if needed, either globally or on a per-location basis.

[7] Free and Open Source Software
[8] Linux Apache MySQL PHP [113, 97, 99]

Packages

Together with repositories, we will introduce the concept of *packages*. Packages in the Aeolus level correspond to the software packages coming from package-based software distributions (like Linux distributions, see [35]).

Definition Each package belongs to a single repository. Packages can be installed on locations, as long as the location has a matching repository: if package belongs to a repository r it can be only installed on locations which have repository r.

Inside each repository packages can have *dependencies* and *conflicts* between themselves. Our dependencies and conflicts take a similar form to these used in the CUDF model [116] (expressive enough to encode Debian package distributions meta-data [65]). Each package might have thus two "properties":

- **Dependencies**: a conjunction of disjunction of packages on which given package depends.

  Semantics: At least one package from each disjunction must be installed on the same location in order to satisfy the dependency (see property *abundance* in [32]).

- **Conflicts**: a list of packages that given package is in conflict with.

  Semantics: None of the packages from this list may be installed on the same location in order to avoid conflict (see property *peace* in [32]).

The package dependencies and conflicts are for us always local to a repository (i.e. a package cannot depend or be in conflict with packages from another repository) and they work in a scope of a given location.

Packages and components As we mentioned before, packages are the layer linking directly the repositories with the components. When we say that a component is *implemented* by a package, it represents a situation where a given service is realized by software provided in a given package.

Each component type can be possibly implemented by many different packages coming from different repositories. For example, a generic mail server may be implemented using various different mail server packages (e.g. Sendmail [107], Postfix [118], Exim [58]), each of them available in many different Linux distributions.

In order for a component of a certain type to be installed on a given location, one of the packages implementing that component must be installed on that location.

Purpose of packages In our model, packages serve several purposes:

1. Packages let us define which components can be installed on which operating systems. We need at least one package *implementing* a given component type present in a repository in order to make this component installable on a machine with this repository.

2. Packages let us model the local incompatibilities between components. If the packages implementing two different component types are not installable together in a certain repository, then these components are in local conflict and cannot be put together on the same machine using that repository, and if they both need to be used, they will need to be deployed on different machines.

Also, incorporating packages in our model should allow us to know exactly which real software packages must be present on a given machine in order to make all the components work. But here we are faced with a trade-off:

- Adding all the meta-data describing packages and their relationships to our model would allow us to know exactly *all packages* that need to be present on a machine. However, if we do it the model would become unmanageable in practice: Debian stable distribution has more than 40.000 packages and hundreds of thousands of dependencies and conflicts, and for each machine we would need to manage such a huge and complex package repository. The size and complexity of the corresponding constraint problem would be enormous.

- Using the `coinst` tool [25] we can reduce the size of the involved package repositories (and thus the constraint problem) by several orders of magnitude. We do it by grouping packages together and using only the information needed to describe the incompatibilities among them (this approach is described in detail in section 7.5.3). This allows us to guarantee that in every case all the necessary packages (i.e. those needed by the components which are placed on a given machine) *can* be actually installed together, but we do not know exactly *how many* packages nor exactly *which of them* will be installed at the end.

In practice we have decided to follow the second approach. The repercussions of this choice will be discussed in section 7.5.3.

Semi-formal definition

The repositories and packages are the first model extension relating our components to locations. Let us then define the repository and package concept semi-formally at the model level:

- Every universe contains now a set of *repositories*.
- Each *repository*:
  - has an unique *name*,
  - contains a set of *packages*.
- Each *package* in a given repository:
  - has an unique *name*,
  - can *depend* on other packages in its repository (this dependency is expressed in form of a conjunction of disjunctions of package names),
  - can be in *conflict* with other packages in its repository (this conflict is expressed in form of a set of package names).
- Component types in a universe are associated with the packages from the repositories by an *implementation* relation.

  For each component type this relation is expressed in form of a set of package names (more exactly, pairs or repository name and package name, in order to identify each package unambiguously).
- Each location in a configuration:
  - has exactly one repository name associated with it (representing the operating system it runs),
  - has a set of package names associated with it (representing the packages installed on the corresponding machine).

The notion of configuration validity with respect to a universe is extended by several new conditions:

- Each location in the configuration must be associated with exactly one repository coming from the universe.
- All the packages on each machine must belong to the repository installed on that machine. Also all the dependencies of each installed package must be fulfilled and there may be no conflicts present (see property *peace* and *abundance* in [32]).
- If a component of a certain type is placed on a given location, then at least one of the packages implementing this component must be installed on that location.

6.4.3 Resources

After introducing the repositories and packages, we will add the concept of *resources* to our formal model. This permits us to address the second part of the component placement problem (installability and coinstallability of components on machines): this time we focus on how the components' need for computing resources is fulfilled.

Intuitively, the model-level resources will represent to the computing resources (e.g. CPU power, RAM memory) of machines in our distributed system: our locations will thus provide them and components will require and use them.

Definition

Basically, each resource in the model corresponds to a location's capacity or a component's need in terms of a certain measurable "computing resource". Each location can provide a some amount (expressed as a natural number) of each resource and components installed on a location consume these resources.

The whole point of introducing resources to our model is to be able to assure that there is enough of them everywhere: on every location in the configuration there must be least as much of each resource provided (by the location itself) than consumed in total (by all the components installed there).

Interpretation

What we understand by resources is really quite abstract and model-dependent. We should remember that the most important thing is their practical effect: resources help us restrict which components can be installed and / or coinstalled together on which machines. The main purpose of such restrictions is to make sure that all the machines will be able handle the load caused by the services operating on them.

So in fact by attributing a certain resource consumption to a given component we do not really mean that this component uses *exactly* this amount of this resource. Instead we rather want to set a kind of deployment policy which says "for the purpose of machine capacity and co-location with other components, suppose that this component needs this amount of computing resources in order to perform at the expected level". Where the quality of service associated with the "expected level" may depend on our deployment policy on a particular system, so the actual consumption attributed to a component should take into account a specificity of its use in a given distributed system.

Virtual resources

Note: we can also attempt to encode other desired system proprieties using "virtual resources", which do not correspond well to actual computing resources:

- Simple example: if we define a "component slot" resource and we make every machine provide one "component slot" resource and every component consume one "component slot" resource, we will ensure that we will have at most one component per machine.
- Another example: we might introduce a "bandwidth consumption" resource. Bandwidth consumption should be probably a property of a connection between locations in order to really model it in an appropriate fashion, but maybe modelling it as a simple per-machine property could be of some utility in certain cases.

Semi-formal definition

The resources are the second model extension relating our components to locations. Let us define now the resource concept semi-formally at the model level:

- Every location in the configuration can *provide resources*.

  This is expressed in a form of a function from a set of names (of the resources that it provides) to positive integer values, which express how much of a certain resource the location provides.

- Conversely, every component type in the universe can *consume resources*.

  This is expressed in a similar fashion, only now the values mean correspond to how much of a given resource each component of that type consumes.

Also the notion of configuration validity with respect to a universe is extended by a new condition: we must make sure that on every location in the configuration there is enough of each resource. More precisely: at least as much of a resource must be provided by the location that is consumed in total by all the components placed on that location.

6.4.4 Cost

The last feature that we will add directly to the model is the location cost. Now every location will have a *cost* value attributed.

This value is supposed to represent the cost of using the machine represented by the given location. In case of a virtual machine from a cloud provider it can be the real cost of the machine which we have to pay per hour or per day. In case of "our" physical machine or a virtual machine in our private cloud it may signify its maintenance cost (i.e. cost of using the machine in comparison to not using it) if we can estimate it, or whatever else seems useful.

Purpose

The purpose of introducing the location cost is almost exclusively to use it in the optimization part of our approach. Basically we will try to minimize the total cost of used (not empty) locations in order to minimize the cost of the whole distributed system deployment.

Interpretation

If we take a simple case where we attribute a cost equal one to each location in the configuration, we would consider (and therefore be able to minimize) the number of machines in our distributed system that are used. This may be already an effective optimization criterion in practice.

In more complex cases we could however try to make the resources and cost attributed to each location correspond to actual real running cost or cloud provider offer for such a machine. This way we could attempt to minimize the real (i.e. measured in actual quantity of money in time) cost of our deployment.

Improvement

This approach to the cost optimization is much better than our "minimize the number of components" idea from the previous chapter, as it lets us actually estimate the real cost of the deployment and not only roughly guess it. The reason for this improvement is the fact that now we have the actual sources of cost, the machines, included in our model.

Semi-formal definition

Let us define now the *location cost* semi-formally:

- Every location in a configuration is now assigned a *cost* property, which is a non-negative integer value.
- We define the cost of the whole deployment (i.e. cost associated to a given configuration) as the sum of the costs of all the machines in the configuration which are used (not empty), i.e. have at least one component placed on them.

6.4.5 Extended specification

As we are extending the Aeolus stateless model, which is used to express the input universe as well as the initial and final configurations of the architecture synthesis problem, we will also extend our request language, used to provide the input specification.

Deployment policy

We do not merely want to upgrade the request language to make it match the extended version of the model, we also want to introduce a new feature to our approach: the possibility to express more detailed deployment policies. Therefore our extended specification will have two purposes:

- As before: to express the user request, which basically corresponds to stating **what** do we want to deploy (i.e. providing a partial configuration that usually needs to be extended in order to be rendered valid).
- And additionally: to express some more general deployment policies provided by the user, which corresponds to specifying **how** do we want to deploy the system (and does not have to comply with the "partial configuration" idea any more).

Before, all our "deployment policy" was in some way encoded in various parts of the universe (e.g. require and provide port arities of component types are in most cases not linked directly to the inherent properties of particular services, but rather defining the deployment policy), now we will also use the specification to express it (for example to impose some interesting local properties on a particular set of locations, etc).

Implementation

As we have seen in chapter 5, when applying the constraint-solving approach to the flat version of the Aeolus stateless model, the fact that we need to encode all the properties imposed on the final configuration in form of a constraint problem and then to be able to reconstruct the configuration from the solution of this problem is restricting our possibilities in the domain of defining these properties. In other words we should probably only add such features to our request language that we can encode in a reasonable way on the constraint level.

Therefore, differently to what we did before, our new request language is not defined beforehand, but rather derived from the elements available in our constraint problem encoding, in an attempt to make it as expressive and useful as possible and still working seamlessly with our approach in practice.

6.5 Extended stateless Aeolus formal model

In this section we will formally define the *extended stateless Aeolus model*, also known as the Zephyrus model, as it is used in our architecture synthesis tool Zephyrus which will be properly introduced in chapter 7. Most of the elements of this formalization are based on the formalization of the flat stateless Aeolus model (also called here "the flat model"), presented in chapter 5. Hence we will refer to the previous definitions where possible and we will only explain in detail the important extensions. Also, most of the basic notations will stay exactly the same.

Nevertheless, we will put effort into assuring that all the following definitions concerning the Zephyrus model are entirely self-sufficient and not require the flat model definitions in order to work well, even if this means repeating some content.

Notations

- We use following notations concerning **natural numbers**:
  - $\mathbb{N}$ denotes natural numbers with zero,
  - $\mathbb{N}^+$ denotes strictly positive natural numbers,
  - $\mathbb{N}^+_\infty$ denotes $\mathbb{N}^+ \cup \{\infty\}$.
- We define the partial functions as objects called **mappings**:

  Given two sets X and Y, we call a *mapping* $f : X \mapsto Y$ any function f such that its domain $dom(f)$ is a finite subset of X and whose image $f[X]$ is included in Y (i.e. $dom(f) \subseteq X \wedge f[X] \subseteq Y$).

- We define a **tuple lookup** operation (which is a meta-notation) as follows:

  Given a tuple $T = \langle \ell_1, \dots \ell_i \rangle$, we note $T.\ell_i$ the lookup operation that retrieves the element ℓ_i from the tuple T.

Names

Before proceeding to formal definitions, we must first introduce several disjoint sets that will be used as unique names to reference various elements of the model. In the following, we suppose given these disjoint infinite (save one) sets:

- a set of port names $\mathcal{P}$, ranged over by p_1, p_2, etc;
- a set of component type names $\mathcal{T}$, ranged over by t_1, t_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 component names $\mathcal{C}$, ranged over by c_1, c_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.

6.5.1 Universe and configuration

Universe

Now we can define the structure of a *component type* and a *universe* in the Zephyrus model. They are similar to the stateless definitions from the previous chapter, but obviously extended with all the new features: *repositories*, *packages* and *resources*.

Component type A *component type* looks almost the same as it did before (flat model equivalent is the definition 10 on page 69), the only difference is that now it also declares the *resource consumption* of its instances:

Definition 26 (Component Type)
A component type *is a quadruple* $\langle \mathbf{P}, \mathbf{R}, \mathbf{C}, \mathbf{f} \rangle$*, where:*

- $\mathbf{P} : \mathcal{P} \mapsto \mathbb{N}_\infty^+$ *is a mapping defining the ports provided by the component type (the domain of the mapping) with their provide arity (the values for each port name);*
- $\mathbf{R} : \mathcal{P} \mapsto \mathbb{N}^+$ *is a mapping defining the ports required by the component type (the domain of the mapping) with their require arity (the values for each port name);*
- $\mathbf{C} \subset \mathcal{P}$ *is the finite set of conflict ports of the component type;*
- $\mathbf{f} : \mathcal{O} \to \mathbb{N}$ *is a function stating how much of each resource the component type consumes.*

We note Γ *the set of all component types.*

Repository and package The *repositories* and *packages* are new concepts, which did not exist in the flat model and were added in the Zephyrus model, therefore this is their first formal definition:

Definition 27 (Package)
A package *is a triple* $\langle \mathbf{R}, \mathbf{C}, \mathbf{f} \rangle$*, where:*

- $\mathbf{R} \subset \wp(\mathcal{K})$[9] *is the set of dependencies of this package (for each set* $\{k_i\} \in \mathbf{R}$ *at least one* k_i *must be installed for this package to be installed as well);*
- $\mathbf{C} \subset \mathcal{K}$ *is the set of packages this package one is in conflict with;*
- $\mathbf{f} : O \to \mathbb{N}$ *is a function stating how much of each resource this package consumes.*

We note Π *the set of all packages.*

Definition 28 (Repository)
A repository $\mathcal{R} : \mathcal{K} \mapsto \Pi$ *is a finite mapping defining the set of available packages (the image of the mapping) with their names (the domain of the mapping). We note* Ω *the set of all repositories.*

Universe Now can formally define what a *universe* is. Before (flat model equivalent is the definition 11 on page 69) it was a very simple entity, basically a mapping from component names to their types. Now it has a much more complex structure, not only declaring the available component types and package repositories, but also bringing them together through the *implementation relation*:

Definition 29 (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 available component types (the image of the mapping) with their names (the domain of the mapping);*
- $\mathbf{I} \subset dom(\mathbf{N}) \times \mathcal{K}$ *is the implementation relation (defining which packages implement which component types);*
- $\mathbf{Y} : \mathcal{D} \mapsto \Omega$ *is a finite mapping defining the set of available repositories (the image of the mapping) with their names (the domain of the mapping);*

To simplify our presentation, we suppose that the repositories of a universe $\mathcal{U}$ *all have distinct domains (i.e. each package name* $k \in \mathcal{K}$ *belongs to at most one of the repositories defined by* $\mathbf{Y}$*).*

Notations

- Given a universe $\mathcal{U} = \langle \mathbf{N}, \mathbf{I}, \mathbf{Y} \rangle$, we note:
  - $\mathcal{U}_{dt}$ the set of names of the component types available in the universe $\mathcal{U}$:

$$\mathcal{U}_{dt} \triangleq dom(\mathcal{U})$$

  - $\mathcal{U}_{dp}$ the set of names of ports used in the universe $\mathcal{U}$:

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

  - $\mathcal{U}_{dr}$ the set of names of repositories in $\mathcal{U}$:

$$\mathcal{U}_{dr} \triangleq dom(\mathbf{Y})$$

[9] We write $\wp(X)$ for the set of subsets of X

– $\mathcal{U}_{dk}$ the set of names of all packages in $\mathcal{U}$:

$$\mathcal{U}_{dk} \triangleq \bigcup_{r \in \mathcal{U}_{dr}} dom(\mathbf{Y}(r))$$

- Moreover:
  - $\mathcal{U}_i : \mathcal{U}_{dt} \mapsto \wp(\mathcal{U}_{dk})$ gives the set of packages implementing each component type in $\mathcal{U}$:

$$\mathcal{U}_i(t) \triangleq \{k \mid \langle t, k \rangle \in \mathbf{I}\}$$

  - $\mathcal{U}_w : \mathcal{U}_{dk} \mapsto \Pi$ gives the mapping from all the package names appearing in $\mathcal{U}$ to the corresponding packages:

$$\mathcal{U}_w \triangleq \bigcup_{r \in \mathcal{U}_{dr}} \mathbf{Y}(r)$$

  - $\mathcal{U}\mathtt{R} : \mathcal{P} \to \wp(\mathcal{U}_{dt})$ gives us all the *requirers* of the given port, i.e. the set of all the component types that belong to the universe $\mathcal{U}$ and require the port in parameter:

$$\mathcal{U}\mathtt{R}(p) \triangleq \{t \mid t \in \mathcal{U}_{dt} \wedge p \in dom(\mathbf{N}(t).\mathbf{R})\}$$

  - $\mathcal{U}\mathtt{P} : \mathcal{P} \to \wp(\mathcal{U}_{dt})$ gives us all the *providers* of the given port, i.e. the set of all the component types that belong to the universe $\mathcal{U}$ and provide the port in parameter:

$$\mathcal{U}\mathtt{P}(p) \triangleq \{t \mid t \in \mathcal{U}_{dt} \wedge p \in dom(\mathbf{N}(t).\mathbf{P})\}$$

  - and $\mathcal{U}\mathtt{C} : \mathcal{P} \to \wp(\mathcal{U}_{dt})$ gives us the *conflicters* of the given port, i.e. the set of all types component types that belong to the universe $\mathcal{U}$ and conflict with the port in parameter (i.e. possess the corresponding conflict port):

$$\mathcal{U}\mathtt{C}(p) \triangleq \{t \mid t \in \mathcal{U}_{dt} \wedge p \in \mathbf{N}(t).\mathbf{C}\}$$

- Also:
  - given a component type name $t \in \mathcal{T}$, we note $\mathcal{U}(t)$ for $\mathbf{N}(t)$,
  - given a repository name $r \in \mathcal{D}$, we note $\mathcal{U}(r)$ for $\mathbf{Y}(r)$,
  - and given a package name $k \in \mathcal{K}$, we note $\mathcal{U}(k)$ for $\mathcal{U}_w(k)$.

$$\mathcal{U}(x) \triangleq \begin{cases} \mathbf{N}(x) & \text{if} \quad x \in \mathcal{T} \\ \mathbf{Y}(x) & \text{if} \quad x \in \mathcal{D} \\ \mathcal{U}_w(x) & \text{if} \quad x \in \mathcal{K} \end{cases}$$

Configuration

Now the time has come to introduce *configurations*, together with *locations* and *components*. The flat configuration (see definition 11 on page 69) declared only the components present in the system and bindings between them. We extend this definition by adding the information about the locations and their properties, as well as including the the placement of components on the locations:

Definition 30 (Configuration)
A configuration *C is a triple $\langle L, W, B \rangle$, where:*

- *L is a mapping from $\mathcal{L}$ to quadruples $\langle \phi, r, M, s \rangle$, 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 names of packages installed on that location;*
  - $s \in \mathbb{N}^+$ *is the cost of using that location;*

- $W : C \mapsto (dom(L) \times \mathcal{T})$ *is a mapping form component names* C *to pairs* $\langle l, t \rangle$*, where* $l \in dom(L)$ *is a location name and* $t \in \mathcal{T}$ *is a component type name, defining the set of names of components which are present in the configuration (the mapping's domain) and stating for each component its location and its type (the mapping's values);*

- $B \subset \mathcal{K} \times dom(W) \times dom(W)$ *is the set of* bindings *existing in the configuration, which are triples consisting of, respectively:*

  - *a port name,*
  - *the name of the component that requires that port*
  - *and the name of the component that provides it.*

Notations

- Given a configuration $C = \langle L, W, B \rangle$, we note:

  - C_l the set of names of all the locations in that configuration;

$$C_l \triangleq dom(L)$$

  - C_c the set of names of all the components in that configuration;

$$C_c \triangleq dom(W)$$

  - C_t the set of names of all the types of components in that configuration;

$$C_t \triangleq \{t \mid \exists c \in C_c,\ \exists l \in C_l,\ W(c) = \langle l, t \rangle\}$$

  - C_k the set of names of all the packages installed in that configuration.

$$C_k \triangleq \bigcup_{l \in C_l} C(l).M$$

- Moreover:

  - Given a location name $l \in \mathcal{L}$, we note $C(l)$ for $L(l)$;
  - Given a component name $c \in \mathcal{C}$, we note $C(c)$ for $W(c)$;

$$C(x) \triangleq \begin{cases} L(x) & \text{if} \quad x \in \mathcal{L} \\ W(x) & \text{if} \quad x \in \mathcal{C} \end{cases}$$

  - Given a component name $c \in \mathcal{C}$, we note $C.\texttt{type}(c)$ for the name of the type of the component c;

$$\forall c \in C_c,\ C.\texttt{type}(c) = t \iff \exists l \in C_l,\ W(c) = \langle l, t \rangle$$

  - Given a location name $l \in C_l$, we note $C.\texttt{cost}(l)$ for the cost of the location l:

$$C.\texttt{cost}(l) \triangleq L(l).s$$

– Given a location name $l \in \mathcal{L}$ and a component type name $t \in \mathcal{T}$, we note $C(l,t)$ the set of components that are placed on the location l and whose type is t;

$$C(l,t) \triangleq \{c \,|\, c \in dom(W) \wedge W(c) = \langle l,t \rangle\}$$

– Given a location name $l \in \mathcal{L}$ and a package name $k \in \mathcal{K}$, we note and $C(l,k)$ the boolean value stating whether the package k is installed on location l.

$$C(l,k) \triangleq \begin{cases} \textbf{true} & \text{if} \quad k \in L(l).M \\ \textbf{false} & \text{if} \quad k \notin L(l).M \end{cases}$$

Configuration validity with respect to a universe

The notion of the configuration validity with respect to a universe becomes more complex than its flat model equivalent (see definition 13 on page 73) because of all the added features. Before we only had to assure that all the components have types coming from the universe and that the bindings between them were formed correctly. Now, with package repositories and resources, there are more properties to check.

Therefore our validity definition will be structured into three separate parts: component-validity, package-validity and resource-validity. These three partial validities will be then put together as a complete configuration validity definition.

First, the component-validity (including in fact to everything what was defined in flat configuration validity) checks the components and bindings present in the configuration, also verifying if all the components are implemented by a package:

Definition 31 (Configuration component-validity w.r.t. universe)
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. $\mathcal{U}$ (noted $\mathcal{U} \vdash_{cmp} C$) if for all the components $c \in C_c$ present in the configuration:*

- *the component's type $t = C.\texttt{type}(c)$ belongs to the universe $\mathcal{U}$;*
  the pair $\langle l,t \rangle = W(c)$ is such that $t \in \mathcal{U}_{dt}$
- *there are no bindings on port p in which the component is the providing side if it does not provide the port p;*

$$\forall p \in \mathcal{P} \setminus dom(\mathcal{U}(t).\mathbf{P}),\ \{c_{req} \,|\, (p, c_{req}, c) \in B\} = \emptyset$$

- *the component's provide ports are not overused;*
  (i.e. the number of components that the component is bound to on port p as a providing component must not be bigger than its provide arity on port p)

$$\forall p \in dom(\mathcal{U}(t).\mathbf{P}),\ \left|\{c_{req} \,|\, (p, c_{req}, c) \in B\}\right| \leq \mathcal{U}(t).\mathbf{P}(p)$$

- *the component's require ports are satisfied;*
  (i.e. the number of components that the component is bound to on port p as a requiring component must be at least as big as its require arity on port p)

$$\forall p \in dom(\mathcal{U}(t).\mathbf{R}),\ \left|\{c_{prov} \,|\, (p, c, c_{prov}) \in B\}\right| \geq \mathcal{U}(t).\mathbf{R}(p)$$

- *there can be no conflicts;*
  (i.e. if a component has a conflict port p then it is the only component in the configuration that can also provide the port p)

$$\forall p \in \mathcal{U}(t).\mathbf{C},\ \left\{c_{confl} \,|\, c_{confl} \in C_c \wedge C(c_{confl}) \in \mathcal{U}\mathrm{P}(p)\right\} \subseteq \{c\}$$

- *and the component is implemented by a package.*

  (i.e. at least one of the packages which implement the component is installed on the same location)

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

Package-validity verifies in turn if the repositories attributed to locations are well defined in the universe and if the packages are correctly installed on the locations throughout the configuration (i.e. if all the dependencies are satisfied and conflicts avoided):

Definition 32 (Configuration package-validity w.r.t. universe)
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. $\mathcal{U}$ (noted $\mathcal{U} \vdash_{pkg} C$) if for all the locations $l \in dom(L)$, the quadruple $\langle \phi, r, M, s \rangle = L(l)$ is such that:*

- *the repository of the location is declared in the universe $\mathcal{U}$;*

$$r \in \mathcal{U}_{dr}$$

- *all the packages installed on the location are declared in the universe $\mathcal{U}$ and belong to the right repository (i.e. the one installed on the location);*

$$M \subseteq \mathcal{U}(r)$$

- *all the dependencies of all the packages installed on the location are satisfied;*

  (i.e. package dependencies are in the CNF[10] form: a conjunction of disjunctions; at least one package from each disjunction must be installed)

$$\forall k \in M,\ \forall d \in \mathcal{U}(k).\mathbf{R},\ \exists k' \in d,\ k' \in M$$

- *and there are no conflicts between the packages installed on the location;*

$$\forall k \in M,\ \mathcal{U}(k).\mathbf{C} \cap M = \emptyset$$

Finally resource-validity is here to check if the resource consumption in the configuration is correct, i.e. if there is no location where more of a certain resource is consumed than provided:

Definition 33 (Configuration resource-validity w.r.t. universe)
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. $\mathcal{U}$ (noted $\mathcal{U} \vdash_{res} C$) if for all locations $l \in dom(L)$ and all resources $o \in O$, the following inequality holds:*

$$\left(\sum_{t \in \mathcal{U}_{dt}} |C(l,t)| \cdot \mathcal{U}(t).\mathbf{f}(o) \right) + \left(\sum_{k \in (L(l).M \cap \mathcal{U}_{dk})} \mathcal{U}(k).\mathbf{f}(k) \right) \leq L(l).\phi(o)$$

In the end we simply put the three precedent definitions into one:

Definition 34 (Configuration validity w.r.t. universe)
A configuration C is valid *w.r.t. a universe $\mathcal{U}$ (noted $\mathcal{U} \vdash C$) if and only if these three conditions hold:*

- *C is component-valid w.r.t. $\mathcal{U}$,*
- *C is package-valid w.r.t. $\mathcal{U}$,*

[10] Conjunctive Normal Form

- *and C is resource-valid w.r.t.* $\mathcal{U}$.

Formally:

$$\mathcal{U} \vdash C \iff \begin{cases} \mathcal{U} \vdash_{cmp} C \\ \mathcal{U} \vdash_{pkg} C \\ \mathcal{U} \vdash_{res} C \end{cases}$$

6.5.2 Specifications

As mentioned before, our extended specification corresponding to the Zephyrus model is not really based on high-level design decisions, but rather designed around finding all the potentially interesting properties which we could easily encode in form of a constraint problem. Let us define now its syntax and semantics.

S	::=	$\mathbf{true} \mid e\ op\ e \mid S \wedge S \mid S \vee S \mid S \Rightarrow S \mid \neg S$		Specification
e	::=	$n \mid X \mid \#\ell \mid e + e \mid e - e \mid n \times e$		Expression
ℓ	::=	$k \mid t \mid p \mid (J_\phi)\{J_r : S_l\}$		Elements
S_l	::=	$\mathbf{true} \mid e_l\ op\ e_l \mid S_l \wedge S_l \mid S_l \vee S_l \mid S_l \Rightarrow S_l \mid \neg S_l$		Local Specification
e_l	::=	$n \mid X \mid \#\ell_l \mid e_l + e_l \mid e_l - e_l \mid n \times e_l$		Local Expression
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	::=	$< \mid \leq \mid = \mid \geq \mid > \mid \neq$		Operators

Table 6.1: Extended Specification Syntax

Syntax

Specifications are defined according the abstract syntax presented in table 6.1. Basically:

- A **specification** S is a set of basic constraints $e\ op\ e$, combined using the usual logical operators $\wedge$, $\vee$, $\Rightarrow$ and $\neg$.
- A **specification expression** e is a numerical expression. It can be either:
  - a **constant** numerical value n;
  - a **free variable** X (those were added for flexibility and are not correlated directly to any part of the final configuration, but rather are supposed to serve as a link between different parts of one specification together if there is need for that);
  - a **cardinality** of an element $\#\ell$, corresponding to how many times a given element ℓ (e.g. instance of a component type) is present in the final configuration;
  - a **sum** or **subtraction** of two expressions, or a **multiplication** by a constant;
- A **specification element** designates something that can be counted using the cardinality expression, the three basic ones are:
  - instances of a **component type** t present,
  - total provided arity of **port** p,
  - number of times **package** k is installed.

- It is also possible to have **constraints on locations**, as locations can be counted through the use of a *cardinality* expression and selected using the term $(J_\phi)\{J_r : S_l\}$, where:
  - **resource constraint** J_ϕ is specifies the resources that must be available on the machine;
  - **repository constraint** J_r is the set of repositories that can be installed on the machine;
  - and the **local specification** S_l is a constraint specifying what are the contents of the machine:
    * **local specification** S_l works exactly the same way as a normal *specification* S, but in local scope of a single machine;
    * the same is true for *local expression* e_l in relation to the normal *expression* e;
    * as well as for the *local element* ℓ_l in relation to the normal *element* ℓ.

Using this syntax we can express some quite elaborate properties, like for example local conflict between two component types t_a and t_b in scope of locations using a certain repository r. This specification:

$$\#(\_)\{r : \#t_a \geq 1 \wedge \#t_b \geq 1\} = 0$$

means:

"there are no machines in the configuration that have repository r installed and contain components of both type t_a and t_b"

Notations

- Suppose given a specification S:
  - $fv(S)$ stands for the set of free variables (i.e. those represented by X in the syntax definition) used in S.

Semantics: configuration validity with respect to a specification

The following definition formally presents the semantics of a specification S in context of determining is a given configuration C satisfies it (for the flat version see definition 14 on page 74):

Definition 35 (Configuration satisfying the specification)
Given a universe $\mathcal{U}$, a configuration C satisfies a specification S (noted $C \vdash S$) if and only if there exists a function $\sigma_X : fv(S) \to \mathbb{Z}$ (assignment of values to the free variables), such that $C, \sigma_X \vdash S$ can be derived from the set of rules presented in tables 6.2 and 6.3.

Basically, these two tables map the number of different elements present in the configuration (e.g. locations with certain properties and content, components of given type, etc.) to the different cardinality expressions $\#\ell$ in the specification, and ensure that the function σ_X (defining values of the free variables) together with this mapping is a solution for S.

6.5.3 Optimization

The optimization criteria that we will define for the Zephyrus model are more complex than their flat model counterparts. As the model is richer, there are many more aspects to take into account when we want to encode an abstract optimization criterion in form of an *optimization function* and an *utility order*. In most cases we will not minimize or maximize a single integer value now, but compare lexicographically whole tuples of values.

However, the most important concepts stay almost the same. *Optimization criteria* are defined exactly as before (see definition 15 on page 77), the only difference is that:

SV:True

$$C, \sigma_X \vdash \textbf{true}$$

SV:Comparison

$$\frac{C, \sigma_X \vdash e_1 \Rightarrow n_1 \qquad C, \sigma_X \vdash e_2 \Rightarrow n_2 \qquad n_1 \; op \; n_2}{C, \sigma_X \vdash e_1 \; op \; e_2}$$

SV:Conjunction

$$\frac{C, \sigma_X \vdash S_1 \qquad C, \sigma_X \vdash S_2}{C, \sigma_X \vdash S_1 \wedge S_2}$$

SV:Disjunction 1

$$\frac{C, \sigma_X \vdash S_1}{C, \sigma_X \vdash S_1 \vee S_2}$$

SV:Disjunction 2

$$\frac{C, \sigma_X \vdash S_2}{C, \sigma_X \vdash S_1 \vee S_2}$$

SV:Implication 1

$$\frac{C, \sigma_X \vdash S_1 \qquad C, \sigma_X \vdash S_2}{C, \sigma_X \vdash S_1 \Rightarrow S_2}$$

SV:Implication 2

$$\frac{C, \sigma_X \nvdash S_1}{C, \sigma_X \vdash S_1 \Rightarrow S_2}$$

SV:Negation

$$\frac{C, \sigma_X \nvdash S}{C, \sigma_X \vdash \neg S}$$

SV:Variable

$$C, \sigma_X \vdash X \Rightarrow \sigma_X(X)$$

SV:Constant

$$C, \sigma_X \vdash n \Rightarrow n$$

SV:Package

$$C, \sigma_X \vdash \#k \Rightarrow \sum_{l \in C_l} | \; C(l).M \cap \{k\} \; |$$

SV:Component Type

$$C, \sigma_X \vdash \#t \Rightarrow \sum_{l \in C_l} | \; C(l, t) \; |$$

SV:Port

$$C, \sigma_X \vdash \#p \Rightarrow \sum_{l \in C_l} \sum_{t \in \mathcal{U}\mathbf{P}(p)} | \; C(l, t) \; | \times \mathcal{U}(t).\mathbf{P}(p)$$

SV:Sum

$$\frac{C, \sigma_X \vdash e_1 \Rightarrow n_1 \qquad C, \sigma_X \vdash e_2 \Rightarrow n_2}{C, \sigma_X \vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

SV:Subtraction

$$\frac{C, \sigma_X \vdash e_1 \Rightarrow n_1 \qquad C, \sigma_X \vdash e_2 \Rightarrow n_2}{C, \sigma_X \vdash e_1 - e_2 \Rightarrow n_1 - n_2}$$

SV:Multiplication

$$\frac{C, \sigma_X \vdash e \Rightarrow n'}{C, \sigma_X \vdash n \times e \Rightarrow n \times n'}$$

SV:Count Locations

$$\frac{L = \left\{ l \in C_l \,|\, (C, l \vDash J_\phi) \wedge (C, l \vDash J_r) \wedge (C, \sigma_X, l \vDash S_l) \right\}}{C, \sigma_X \vdash \#(J_\phi)\{J_r : S_l\} \Rightarrow | \; L \; |}$$

Table 6.2: Specification Validation

SV:Resource Constraint 1

$$C, l \vDash \_$$

SV:Resource Constraint 2

$$\frac{C, l \vDash J_\phi \qquad C(l).\phi(o)\ op\ n}{C, l \vDash o\ op\ n; J_\phi}$$

SV:Repository Constraint 1

$$\frac{C(l).r = r}{C, l \vDash r}$$

SV:Repository Constraint 2

$$\frac{C(l).r = r}{C, l \vDash r \vee J_r}$$

SV:Repository Constraint 3

$$\frac{C, l \vDash J_r}{C, l \vDash r \vee J_r}$$

SV:Local:True

$$C, \sigma_X, l \vDash \textbf{true}$$

SV:Local:Comparison

$$\frac{C, \sigma_X, l \vDash e_{l_1} \Rightarrow n_1 \qquad C, \sigma_X, l \vDash e_{l_2} \Rightarrow n_2 \qquad n_1\ op\ n_2}{C, \sigma_X, l \vDash e_{l_1}\ op\ e_{l_2}}$$

SV:Local:Conjunction

$$\frac{C, \sigma_X, l \vDash S_{l_1} \qquad C, \sigma_X, l \vDash S_{l_2}}{C, \sigma_X, l \vDash S_{l_1} \wedge S_{l_2}}$$

SV:Local:Disjunction 1

$$\frac{C, \sigma_X, l \vDash S_{l_1}}{C, \sigma_X, l \vDash S_{l_1} \vee S_{l_2}}$$

SV:Local:Disjunction 2

$$\frac{C, \sigma_X, l \vDash S_{l_2}}{C, \sigma_X, l \vDash S_1 \vee S_{l_2}}$$

SV:Local:Implication 1

$$\frac{C, \sigma_X, l \vDash S_{l_1} \qquad C, \sigma_X, l \vDash S_{l_2}}{C, \sigma_X, l \vDash S_{l_1} \Rightarrow S_{l_2}}$$

SV:Local:Implication 2

$$\frac{C, \sigma_X, l \nvDash S_{l_1}}{C, \sigma_X, l \vDash S_{l_1} \Rightarrow S_{l_2}}$$

SV:Local:Negation

$$\frac{C, \sigma_X, l \nvDash S_l}{C, \sigma_X, l \vDash \neg S_l}$$

SV:Local:Variable

$$C, \sigma_X, l \vDash X \Rightarrow \sigma_X(X)$$

SV:Local:Constant

$$C, \sigma_X, l \vDash n \Rightarrow n$$

SV:Local:Package

$$C, \sigma_X, l \vDash \#k \Rightarrow \mid C(l).M \cap \{k\} \mid$$

SV:Local:Component Type

$$C, \sigma_X, l \vDash \#t \Rightarrow \mid C(l, t) \mid$$

SV:Local:Port

$$C, \sigma_X, l \vDash \#p \Rightarrow \sum_{t \in \mathcal{U}\mathbf{P}(p)} \mid C(l, t) \mid \times\, \mathcal{U}(t).\mathbf{P}(p)$$

SV:Local:Sum

$$\frac{C, \sigma_X, l \vDash e_{l_1} \Rightarrow n_1 \qquad C, \sigma_X, l \vDash e_{l_2} \Rightarrow n_2}{C, \sigma_X, l \vDash e_{l_1} + e_{l_2} \Rightarrow n_1 + n_2}$$

SV:Local:Subtraction

$$\frac{C, \sigma_X, l \vDash e_{l_1} \Rightarrow n_1 \qquad C, \sigma_X, l \vDash e_{l_2} \Rightarrow n_2}{C, \sigma_X, l \vDash e_{l_1} - e_{l_2} \Rightarrow n_1 - n_2}$$

SV:Local:Multiplication

$$\frac{C, \sigma_X, l \vDash e_l \Rightarrow n'}{C, \sigma_X, l \vDash n \times e_l \Rightarrow n \times n'}$$

Table 6.3: Local Specification Validation

- now our *optimization functions* f_{opt} take configurations of Zephyrus model (and not the flat model),
- and the preorder $\left(\Theta, \leq_{opt}\right)$ is now a relation on configurations of Zephyrus model (and not the flat model).

Let us now see how can we adapt our two favourite optimization criteria defined before (*compact* and *conservative*) to the new circumstances.

Compact

Let us recall, that the principal idea of the *compact* optimization criterion was to minimize the approximate cost of deploying our final configuration. Before, we were doing that in a rather crude way, by minimizing the number of components present in the configuration. Now we have a much more precise tool to accomplish this objective: location cost. In the new version of the *compact* optimization criterion we will thus first of all try to minimize the cost of all the locations that we use (i.e. which have at least one component placed on them in the final configuration).

But if we do only that, we may end up with some superfluous components present in our final configurations. If we compare only the total location cost, we are simply not able to recognize the difference between two configurations which use the same set of locations, e.g. one that is actually optimal and one that has some extra unnecessary components placed additionally on these locations. Putting something that serves no purpose in our configuration does not seem to be a very elegant solution if we want to be *compact*, thus after minimizing the total location cost we will lexicographically minimize the number of components present in the whole configuration. And, for a good measure, in the end (in the lexicographic sense) we will also minimize the number of packages installed.

Our optimization function $compact : \Theta \to (\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})$ will thus take the following form:

$$compact(C) = \langle a_1, a_2, a_3 \rangle \quad \text{where:}$$

$$\begin{cases} a_1 = \sum_{l \in C_l} \begin{cases} C.\texttt{cost}(l) & \text{if} \quad \exists_{t \in C_t} |C(l,t)| \geq 1 \\ 0 & \text{otherwise} \end{cases} \\ a_2 = |C_c| \\ a_3 = \sum_{l \in C_l} \sum_{k \in C_k} \begin{cases} 1 & \text{if} \quad C(l,k) \\ 0 & \text{otherwise} \end{cases} \end{cases}$$

These three three sub-criteria will be compared lexicographically: only if the value corresponding to the first criterion (total location cost) is minimal, then we will move on to minimizing the second value (number of components) and finally the third one (number of packages). Our utility order $\left((\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}), \leq_{opt}\right)$ is thus the following:

$$\forall a_1, a_2, a_3, b_1, b_2, b_3 \in \mathbb{Z} \qquad \langle a_1, a_2, a_3 \rangle \leq_{opt} \langle b_1, b_2, b_3 \rangle \iff \bigvee \begin{cases} a_1 < b_1 \\ (a_1 = b_1 \wedge a_2 < b_2) \\ (a_1 = b_1 \wedge a_2 = b_2 \wedge a_2 \leq b_2) \end{cases}$$

Conservative

The goal of the *conservative* optimization criterion was to minimize the difference between the initial and the final configuration. We were doing that by counting and minimizing the number of components which have been added, removed, or changed their type. Now we can do almost the same thing, but taking into account the component placement: only components which keep their type **and** stay on the same location will be counted as unchanged. Additionally we will also count the packages which have been installed or uninstalled on each location.

Moreover, in order to consider all the dimensions of the problem, we will perform two other optimizations afterwards (lexicographically): among the configurations with the minimal the number of component and package differences we will choose the ones which have the minimal total cost of used locations and finally, as in the previous case, we will minimize the number of packages installed throughout our configuration.

Our optimization function $conservative : \Theta \to (\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z})$ will thus take the following form:

$$conservative(C) = \langle a_1, a_2, a_3 \rangle \quad \text{where:}$$

$$\begin{cases} a_1 = \left\{ \begin{array}{l} \sum_{c \in C} \left\{ \begin{array}{lll} 1 & \text{if} & c \in dom(W_{init}) \wedge c \notin dom(W_{fin}) \quad \text{(case 1)} \\ 1 & \text{if} & c \notin dom(W_{init}) \wedge c \in dom(W_{fin}) \quad \text{(case 2)} \\ 1 & \text{if} & c \in dom(W_{init}) \wedge c \in dom(W_{fin}) \wedge W_{init}(c) \neq W_{fin}(c) \quad \text{(case 3)} \\ 0 & \multicolumn{2}{l}{otherwise \quad \text{(case 4)}} \end{array} \right. \\ \quad + \\ \sum_{\substack{l \in \mathcal{L} \\ k \in \mathcal{K}}} \left\{ \begin{array}{lll} 1 & \text{if} & C_{init}(l,k) \neq C_{fin}(l,k) \quad \text{(case 5)} \\ 0 & \multicolumn{2}{l}{otherwise \quad \text{(case 6)}} \end{array} \right. \end{array} \right. \\ a_2 = \sum_{l \in C_l} \left\{ \begin{array}{ll} C.\texttt{cost}(l) & \text{if} \quad \exists_{t \in C_t} |C(l,t)| \geq 1 \\ 0 & \text{otherwise} \end{array} \right. \\ a_3 = \sum_{\substack{l \in C_l \\ k \in C_k}} \left\{ \begin{array}{ll} 1 & \text{if} \quad C(l,k) \\ 0 & \text{otherwise} \end{array} \right. \end{cases}$$

As we can see, we count the component differences almost exactly as before:

- we take each component name $c \in C$ and if the component with this name is either:
  - present only in the initial configuration, (case 1)
  - or present only in the final configuration, (case 2)
  - or is present in both configurations, but changes its type or location, (case 3)

  we count it for one difference;
- components which:
  - are absent from both configurations
  - or are present in both, but have the same type and are on the same location,

  obviously do not count as a difference. (case 4)

In addition, we also count the package differences:

- we take each location name $l \in \mathcal{L}$ and each package name $k \in \mathcal{K}$ and if:
  - this package was installed on that location in the initial configuration and is not installed there in the final one,
  - or, symmetrically, it was not installed on that location in the initial configuration and is installed there in the final one,

  we count it for one difference; (case 5)
- packages which do not change their installation status do not count as differences. (case 6)

Of course for the *conservative* optimization criterion we use exactly the same utility order as for *compact*: lexicographic minimization of the elements of a tuple (in this case: a triple).

6.5.4 Conclusion

After setting up the whole framework for the extended architecture synthesis problem we can finally state it in a formal way. The following definitions are almost exactly the same as the corresponding flat model definitions (definition 36 on page 132 and definition 37 on page 132), but their meaning is different, as now they are made in the context of the full Zephyrus model (e.g. when we write "universe" here, we reference the universe concept as it was defined in this chapter).

Definition 36 (Extended architecture synthesis problem)
Given

- *an initial configuration* C_{init},
- *a universe* $\mathcal{U}$
- *and a specification* S,

find a final configuration C *such that*

- C *is valid w.r.t.* $\mathcal{U}$ *(see definition 13 on page 73)*
- *and* C *satisfies* S *(see definition 14 on page 74).*

Definition 37 (Optimal solution of the extended architecture synthesis problem)
Given architecture synthesis problem inputs:

- *an initial configuration* C_{init},
- *a universe* $\mathcal{U}$,
- *a specification* S

and given formalized optimization criteria *(see definition 15 on page 77):*

- *an optimization function* $f_{opt} : \Theta \rightarrow V_{opt}$,
- *with an utility order* $\left(V_{opt}, \leq_{opt}\right)$

which together induce a total preorder on configurations $\left(\Theta, \leq_{opt}\right)$,

an optimal solution *of this architecture synthesis problem according to the provided optimization criteria is a final configuration* C_{fin}^{opt} *which is every maximal element among all the correct solutions of this problem in the preorder* $\leq_{opt}$*, i.e. for each final configuration* C_{fin} *which is a correct solution of the problem (see definition 16 on page 78),* C_{fin}^{opt} *must be* not worse *than* C_{fin} *according to the preorder* $\leq_{opt}$*:*

$$C_{fin} \leq_{opt} C_{fin}^{opt}$$

6.6 Adapting the constraint solving approach

After specifying formally all the parts of the Zephyrus model and defining the extended architecture synthesis problem, let us now proceed to adapt our constraint-based approach in order to make it work for solving this new version of the problem.

As the Zephyrus model is almost a strict extension of the flat model, the new approach will be also an extension of the previous one. We will basically:

- extend the constraint problem definitions with multi-objective optimization in order to adapt it to the lexicographic optimization criteria,
- add new variables and constraints to the constraint encoding of the architecture synthesis problem,
- extend the final configuration generation in order to make it take into account the locations and component placement.

6.6.1 Constraint problem

The constraint problem syntax and semantics will stay basically the same in the new approach. Following tables and definitions still hold, and thus will not be repeated:

- **constraint syntax** provided in table 5.1,
- **constraint semantics** provided in table 5.2,
- **constraint problem** defined in definition 18 on page 82,
- **constraint problem solution** defined in definition 19 on page 82.
- **utility order** for optimization criteria $\max(e)$ and $\min(e)$ defined in definition 20 on page 83
- **constraint problem optimal solution** defined in definition 21 on page 83.

We will only have to extend slightly the optimization criteria syntax and add one definition in order to introduce the multi-objective optimization to our constraint problems and be able to support the refined optimization criteria.

opt_s	::=	$\max(e) \mid \min(e)$	Single-Objective Optimization Criterion
opt_m	::=	$\text{lex}(opt_s, opt_s, \ldots, opt_s)$	Multi-Objective Optimization Criterion
opt	::=	$opt_s \mid opt_m$	Optimization Criterion

Table 6.4: Extended constraint problem optimization criteria syntax

Table 6.4 presents the new formal syntax of our optimization criteria. Basically an optimization criterion *opt* is now either:

- to perform a single-objective optimization:
  - to `maximize` the expression e,
  - or to `maximize` the expression e;
- or to apply a multi-objective optimization, optimizing lexicographically a non-empty list of single-objective optimization criteria.

As we have already covered the utility orders corresponding to the single-objective optimization criteria (in definition 20 on page 83), now we only need to specify the utility order corresponding to the lexicographic optimization criterion and we will be able to define the optimal solutions of the constraint problems according to multi-objective criteria (see definition 21 on page 83):

Definition 38 (Lexicographic optimization criterion utility)
A lexicographic optimization criterion $opt = lex(opt_1, opt_2, \ldots, opt_k)$ *induces the* utility order $\leq_{opt}$ *on solutions (i.e. variable mappings) in the following way:*

We take k utility orders $\langle \leq_1, \leq_2, \ldots, \leq_k \rangle$ *induced by our single-objective optimization criteria, where order* $\leq_i$ *is induced by the optimization criterion* opt_i*. The utility order* $\leq_{opt}$ *corresponds to dictionary ordering using these k orders, i.e. we compare two solutions* σ_A *and* σ_B *using the first order* $\leq_1$*, if they are equal then we use* $\leq_2$*, and so on, and the first order in which they are not equal determines their relationship in the order* $\leq_{opt}$*. Formally:*

$$\sigma_A \leq_{opt} \sigma_B \iff \exists_{m \geq 1} \left(\forall_{i<m} \quad \sigma_A \leq_i \sigma_B \quad \wedge \quad \sigma_B \leq_i \sigma_A \right) \wedge (\sigma_A \leq_m \sigma_B)$$

6.6.2 Encoding

Encoding the extended architecture synthesis problem into constraint problem form follows the same basic principles as we have already established for the flat model. As the whole structure of this encoding and all the fundamental ideas are already well grounded, let us proceed to present all the variables and constraints that are involved in the encoding and explain their role.

Variables

- The **global** cardinality variables $N(\ell)$:
  - The **global component type** variables: $N(t)$
    For each component type t present in the input universe we have a separate integer variable $N(t)$, whose value corresponds to the number of instances of this component type t which are present in the whole final configuration.
  - The **global port** variables: $N(p)$
    For each port p mentioned in the input universe we have a separate integer variable $N(p)$, whose value corresponds to the the total arity of this port p provided by all the components present in the whole final configuration (without differentiating between ports that are bound and unbound).
  - The **global package** variables: $N(k)$
    For each package k available in any repository of the input universe we have a separate integer variable $N(k)$, whose value corresponds to the number of times this package k is installed in the whole final configuration (or, in other words, on how many locations it is installed, as a package cannot be installed multiple times on a single location).
- The **local** cardinality variables $N(l, \ell_l)$:
  - The **local component type** variables: $N(l, t)$
    For each pair of a component type t present in the input universe and a location l available in the input configuration we have a separate integer variable $N(l, t)$, whose value corresponds to the number of instances of this component type t placed on this location l in the final configuration.
  - The **local port** variables: $N(l, p)$
    For each pair of a port p mentioned in the input universe and a location l available in the input configuration we have a separate integer variable $N(l, p)$, whose value corresponds to the total arity of this port p provided by all the components placed on this location l in the final configuration (without differentiating between ports that are bound and unbound).
  - The **local package** variables: $N(k)$
    For each package k available in any repository of the input universe and a location l available in the input configuration we have a separate **boolean**[11] variable $N(l, k)$, whose value is equal 1 if this package k is installed on the location l and 0 otherwise.
- The **repository** variables: $R(l, r)$
  For each repository r available in the input universe and a location l available in the input configuration we have a separate **boolean** variable $R(l, r)$, whose value is equal 1 if repository r is attributed to the location l and 0 otherwise.
- The **binding** variables: $B(p, t_r, t_p)$
  For each possible kind of binding between the components of our final configuration: the port p, the providing component's type t_p and the requiring component's type t_r; we have a separate integer variable $B(p, t_r, t_p)$, whose value corresponds to the number of bindings of this kind existing in the final configuration.

[11] In our encoding boolean variables are in fact integer variables with domains restricted to $\{0, 1\}$.

Let us define formally the set of variables $\mathcal{V}$ and the function attributing them their domains $\mathcal{D}$ of a constraint problem $\mathrm{P} = \langle \mathcal{V}, \mathcal{D}, C \rangle$, corresponding to an extended architecture synthesis problem with an input universe $\mathcal{U}$ and an initial configuration C^{init}:

$$\mathcal{V} = \bigcup \left\{ \begin{array}{c} \bigcup\limits_{t \in \mathcal{U}_{dt}} N(t) \quad \cup \quad \bigcup\limits_{p \in \mathcal{U}_{dp}} N(p) \quad \cup \quad \bigcup\limits_{k \in \mathcal{U}_{dk}} N(k) \\ \bigcup\limits_{l \in C_l^{init}} \left(\bigcup\limits_{t \in \mathcal{U}_{dt}} N(l,t) \quad \cup \quad \bigcup\limits_{p \in \mathcal{U}_{dp}} N(l,p) \quad \cup \quad \bigcup\limits_{k \in \mathcal{U}_{dk}} N(l,k) \quad \cup \quad \bigcup\limits_{r \in \mathcal{U}_{dr}} R(l,r) \right) \\ \bigcup\limits_{p \in \mathcal{U}_{dp}} \bigcup\limits_{t_r \in \mathcal{U}_{\mathrm{R}}(p)} \bigcup\limits_{t_p \in \mathcal{U}_{\mathrm{R}}(p)} B(p, t_r, t_p) \end{array} \right.$$

$$\forall v \in \mathcal{V}, \quad \mathcal{D}(v) = \begin{cases} \{0,1\} & \textit{if} \quad v \in \left(\bigcup\limits_{k \in \mathcal{U}_{dk}} N(k) \right) \cup \left(\bigcup\limits_{l \in C_l^{init}} \bigcup\limits_{r \in \mathcal{U}_{dr}} R(l,r) \right) \\ \mathbb{N} & \textit{otherwise} \end{cases}$$

Table 6.5: Extended encoding: variables.

Constraints

Universe validity We present in table 6.6 the whole encoding of the rules of validity of a configuration C with respect to a universe $\mathcal{U}$ into a constraint C. It is structured in four parts:

- **Coherence**

  equations 6.1 contain constraints what we which are not linked directly to the question of configuration validity, but are in fact establishing a basic **coherence** among the variables of a constraint problem:

  - first three lines encode the **distribution** of components, ports and packages on the locations, linking the global variables for each kind of element with the local ones: for every element ℓ (i.e. component type, port or package), the total number of instances of ℓ (i.e. the value of $N(\ell)$) in the configuration is the sum of its instances on each location (i.e. the sum of all $N(l, \ell)$ for all the locations l);

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

  - the fourth line states that the number of times port p is provided on a location l is equal to the total provide arity of port p of all the components placed on location l.

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

- equations 6.2 correspond to the **component-validity** of the configuration (see definition 31 on page 124). They contain all the rules inherited from the constraint encoding for the flat model, so basically all what concerns the bindings between component types, and additionally the rule assuring the correct implementation of components with packages. Let us recall:

$$\begin{cases} \bigwedge\limits_{t\in\mathcal{U}_{dt}} & N(t) = \sum\limits_{l\in C_l} N(l,t) \\ \bigwedge\limits_{k\in\mathcal{U}_{dk}} & N(k) = \sum\limits_{l\in C_l} N(l,k) \\ \bigwedge\limits_{p\in\mathcal{U}_{dp}} & N(p) = \sum\limits_{l\in C_l} N(l,p) \\ \bigwedge\limits_{l\in C_l}\bigwedge\limits_{p\in\mathcal{U}_{dp}} & N(l,p) = \sum\limits_{t_p\in\mathcal{U}\mathrm{P}(p)} \mathcal{U}(t_p).\mathbf{P}(p)\times N(l,t_p) \end{cases} \tag{6.1}$$

$$\begin{cases} \bigwedge\limits_{p\in\mathcal{U}_{dp}}\bigwedge\limits_{t_r\in\mathcal{U}\mathrm{R}(p)} \mathcal{U}(t_r).\mathbf{R}(p)\times N(t_r) \le \sum\limits_{t_p\in\mathcal{U}\mathrm{P}(p)} B(p,t_p,t_r) \\ \bigwedge\limits_{p\in\mathcal{U}_{dp}} \begin{cases} \bigwedge\limits_{t_p\in\{t\in\mathcal{U}\mathrm{P}(p)\mid\mathcal{U}(t).\mathbf{P}(p)<\infty\}} \mathcal{U}(t_p).\mathbf{P}(p)\times N(t_p) \ge \sum\limits_{t_r\in\mathcal{U}\mathrm{R}(p)} B(p,t_p,t_r) \\ \bigwedge\limits_{t_p\in\{t\in\mathcal{U}\mathrm{P}(p)\mid\mathcal{U}(t).\mathbf{P}(p)=\infty\}} N(t_p)=0 \Rightarrow \sum\limits_{t_r\in\mathcal{U}\mathrm{R}(p)} B(p,t_p,t_r)=0 \end{cases} \\ \bigwedge\limits_{p\in\mathcal{U}_{dp}}\bigwedge\limits_{t_r\in\mathcal{U}\mathrm{R}(p)}\bigwedge\limits_{t_p\in\mathcal{U}\mathrm{P}(p)} B(p,t_p,t_r) \le N(t_r)\times N(t_p) \\ \bigwedge\limits_{p\in\mathcal{U}_{dp}} \begin{cases} \bigwedge\limits_{t\in\mathcal{U}\mathrm{C}(p)\cap\mathcal{U}\mathrm{P}(p)} N(t)\le 1 \\ \bigwedge\limits_{t\in\mathcal{U}\mathrm{C}(p)\cap\mathcal{U}\mathrm{P}(p)} N(t)=1 \Rightarrow \bigwedge\limits_{t'\in\mathcal{U}\mathrm{P}(p),t'\neq t} N(t')=0 \\ \bigwedge\limits_{t\in\mathcal{U}\mathrm{C}(p)\setminus\mathcal{U}\mathrm{P}(p)} N(t)>0 \Rightarrow \bigwedge\limits_{t'\in\mathcal{U}\mathrm{P}(p)} N(t')=0 \end{cases} \\ \bigwedge\limits_{t\in\mathcal{U}_{dt}} N(l,t)\ge 1 \Rightarrow \sum\limits_{k\in\mathcal{U}_i(t)} N(l,k)\ge 1 \end{cases} \tag{6.2}$$

$$\bigwedge_{l\in C_l} \begin{cases} \sum\limits_{r\in\mathcal{U}_{dr}} R(l,r)=1 \\ \bigwedge\limits_{r\in\mathcal{U}_{dr}} R(l,r)=1 \Rightarrow \begin{cases} \bigwedge\limits_{k\in\mathcal{U}(r)} N(l,k)\le 1 \\ \bigwedge\limits_{k\in\mathcal{U}_{dk}\setminus\mathcal{U}(r)} N(l,k)=0 \end{cases} \\ \bigwedge\limits_{k_1\in\mathcal{U}_{dk}}\bigwedge\limits_{K\in\mathcal{U}(k_1).\mathbf{R}} N(l,k_1)\le\sum\limits_{k_2\in K} N(l,k_2) \\ \bigwedge\limits_{k_1\in\mathcal{U}_{dk}}\bigwedge\limits_{k_2\in\mathcal{U}(k_1).\mathbf{C}} N(l,k_1)+N(l,k_2)\le 1 \end{cases} \tag{6.3}$$

$$\bigwedge_{l\in C_l}\bigwedge_{o\in O} \sum_{x\in\mathcal{U}_{dt}\cup\mathcal{U}_{dk}} \mathcal{U}(x).\mathbf{f}(o)\times N(x,l) \;\le\; C(l).\phi(o) \tag{6.4}$$

Table 6.6: Extended encoding: universe validity constraints

– the first line states that all the **requirements** of a component on all the ports must be satisfied by enough bindings on these ports (each with a different providing component);

$$\bigwedge_{p \in \mathcal{U}_{dp}} \bigwedge_{t_r \in \mathcal{U}_{\mathrm{R}}(p)} \mathcal{U}(t_r).\mathbf{R}(p) \times N(t_r) \leq \sum_{t_p \in \mathcal{U}_{\mathrm{P}}(p)} B(p, t_p, t_r)$$

– the second and third line state that no component cannot **provide** more bindings on a given port than their provide arity (the second line handles the finite provide arities and the third one handles infinite provide arities);

$$\bigwedge_{p \in \mathcal{U}_{dp}} \begin{cases} \bigwedge\limits_{t_p \in \{t \in \mathcal{U}_{\mathrm{P}}(p) | \mathcal{U}(t).\mathbf{P}(p) < \infty\}} \mathcal{U}(t_p).\mathbf{P}(p) \times N(t_p) \geq \sum\limits_{t_r \in \mathcal{U}_{\mathrm{R}}(p)} B(p, t_p, t_r) \\ \\ \bigwedge\limits_{t_p \in \{t \in \mathcal{U}_{\mathrm{P}}(p) | \mathcal{U}(t).\mathbf{P}(p) = \infty\}} N(t_p) = 0 \Rightarrow \sum\limits_{t_r \in \mathcal{U}_{\mathrm{R}}(p)} B(p, t_p, t_r) = 0 \end{cases}$$

– the fourth line states that there should not be two bindings on a given port between the same two components (i.e. the **binding unicity** principle);

$$\bigwedge_{p \in \mathcal{U}_{dp}} \bigwedge_{t_r \in \mathcal{U}_{\mathrm{R}}(p)} \bigwedge_{t_p \in \mathcal{U}_{\mathrm{P}}(p)} B(p, t_p, t_r) \leq N(t_r) \times N(t_p)$$

– the lines fifth, sixth and seventh ensure that when a component is in **conflict** with a port, there is no other component providing that port in the configuration (the fifth and sixth line handles components that are in conflict with themselves and the seventh line handles all the other components);

$$\bigwedge_{p \in \mathcal{U}_{dp}} \begin{cases} \bigwedge\limits_{t \in \mathcal{U}_{\mathrm{C}}(p) \cap \mathcal{U}_{\mathrm{P}}(p)} N(t) \leq 1 \\ \bigwedge\limits_{t \in \mathcal{U}_{\mathrm{C}}(p) \cap \mathcal{U}_{\mathrm{P}}(p)} N(t) = 1 \Rightarrow \bigwedge\limits_{t' \in \mathcal{U}_{\mathrm{P}}(p), t' \neq t} N(t') = 0 \\ \bigwedge\limits_{t \in \mathcal{U}_{\mathrm{C}}(p) \setminus \mathcal{U}_{\mathrm{P}}(p)} N(t) > 0 \Rightarrow \bigwedge\limits_{t' \in \mathcal{U}_{\mathrm{P}}(p)} N(t') = 0 \end{cases}$$

– and finally the seventh line encodes the **implementation** relation between component types and packages: if a component of type t is installed on a location l, then at least one its implementing packages $k \in \mathcal{U}_i(t)$ must be installed on that location.

$$\bigwedge_{t \in \mathcal{U}_{dt}} N(l, t) \geq 1 \Rightarrow \sum_{k \in \mathcal{U}_i(t)} N(l, k) \geq 1$$

- equations 6.3 correspond to the **package-validity** of the configuration (see definition 32 on page 125), i.e. the constraints related to repositories and packages:

  – the first line states that exactly one repository must be installed on every location;

$$\bigwedge_{l \in C_l} \sum_{r \in \mathcal{U}_{dr}} R(l, r) = 1$$

  – the second line states when the repository r is installed on a location l, only the packages which belong to that repository can be installed on l;

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

– the third line encodes the dependency relation between packages, which is expressed in form of conjunction of disjunctions of packages: if package k_1 is installed on location l, then at least one package from each disjunction clause of its CNF dependencies $\mathcal{U}(k_1).\mathbf{R}$ must be also installed on location l;

$$\bigwedge_{l \in C_l} \bigwedge_{k_1 \in \mathcal{U}_{dk}} \bigwedge_{K \in \mathcal{U}(k_1).\mathbf{R}} N(l, k_1) \leq \sum_{k_2 \in K} N(l, k_2)$$

– and the fourth line encodes the conflicts between packages: if package k_1 is installed on location l, then none of the packages which are in conflict with it $\mathcal{U}(k_1).\mathbf{C}$ can be installed on location l.

$$\bigwedge_{l \in C_l} \bigwedge_{k_1 \in \mathcal{U}_{dk}} \bigwedge_{k_2 \in \mathcal{U}(k_1).\mathbf{C}} N(l, k_1) + N(l, k_2)$$

- equation 6.4 correspond to the **resource-validity** of the configuration (see definition 33 on page 125), encodes resource usage in each location: on every location l the consumption of each resource o by components and packages cannot exceed the amount of this resource provided on the location l:

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

Satisfying the specification We present in table 6.7 the encoding of validity with respect to a specification S. As now the structure of our request language matches very closely the structure of our constraints, the resulting constraint C is in fact almost identical to S. Only the references to component types, ports and packages, as well as the resources and repository constraints need to be translated into their equivalent in the constraint syntax. The translation is done by induction on the structure of S, and uses statements of the forms:

- $\vdash S : C$ for a **specification** S translated to a **constraint** C,
- $\vdash e : e'$ for a **specification expression** e translated to a **constraint expression** e',
- $l \vdash S_l : C$ for a **local specification** S_l, concerning the **location** l, translated to a **constraint** C,
- $l \vdash e_l : e'_l$ for a **local specification expression** e_l, concerning the **location** l, translated to a **constraint expression** e',
- $l \vdash J_\phi : C$ for **resource constraints** concerning the **location** l,
- $l \vdash J_r : C$ for **repository constraints** concerning the **location** l.

The most interesting rules in our translation are Element Cardinality, Location Cardinality, Local:Element Cardinality, Local:Resource Constraint and Local:Repository Constraint:

- Rule Element Cardinality states that $\#\ell$, which corresponds to the number of instances of element ℓ in the whole configuration, is encoded as the global variable $N(\ell)$.
- Rule Local:Element Cardinality applies when $\#\ell_l$ is used inside a local specification in context of a location l. In that case, $\#\ell_l$ corresponds to the number of instances of ℓ_l present on that location l, and thus is encoded as the local variable $N(l, \ell_l)$.
- Rule Location Cardinality counts the number of locations validating the provided conditions: a resource constraint, a repository constraint and a local specification. To do so, it takes each location l present in the configuration and counts it in, using the reified constraints, only if that location validates all the three constraints.

$$\textsc{True}\quad \vdash \textbf{true} : \textbf{true}$$

$$\textsc{Comparison}\quad \frac{\vdash e_1 : e_1' \qquad \vdash e_2 : e_2'}{\vdash e_1\ op\ e_2 : e_1'\ op\ e_2'}$$

$$\textsc{Negation}\quad \frac{\vdash S : C}{\vdash \neg S : \neg C} \qquad \textsc{Conjunction}\quad \frac{\vdash S_1 : C_1 \qquad \vdash S_2 : C_2}{\vdash S_1 \wedge S_2 : C_1 \wedge C_2} \qquad \textsc{Disjunction}\quad \frac{\vdash S_1 : C_1 \qquad l \vdash S_2 : C_2}{\vdash S_1 \vee S_2 : C_1 \vee C_2} \qquad \textsc{Implication}\quad \frac{\vdash S_1 : C_1 \qquad \vdash S_2 : C_2}{\vdash S_1 \Rightarrow S_2 : C_1 \Rightarrow C_2}$$

$$\textsc{Constant}\quad \vdash n : n \qquad \textsc{Variable}\quad \vdash X : X \qquad \textsc{Element Cardinality}\quad \vdash \#\ell : N(\ell)$$

$$\textsc{Location Cardinality}\quad \frac{l \vdash J_\phi : C_1^l \qquad l \vdash J_r : C_2^l \qquad l \vdash S_l : C_3^l}{\vdash \#(J_\phi)\{J_r : S_l\} : \sum_{l \in C_l} \left\| C_1^l \wedge C_2^l \wedge C_3^l \right\|}$$

$$\textsc{Sum}\quad \frac{\vdash e_1 : e_1' \qquad \vdash e_2 : e_2'}{\vdash e_1 + e_2 : e_1' + e_2'} \qquad \textsc{Subtraction}\quad \frac{\vdash e_1 : e_1' \qquad \vdash e_2 : e_2'}{\vdash e_1 - e_2 : e_1' - e_2'} \qquad \textsc{Multiplication}\quad \frac{\vdash e : u}{\vdash n \times e : n \times u}$$

$$\textsc{Local:True}\quad l \vdash \textbf{true} : \textbf{true}$$

$$\textsc{Local:Comparison}\quad \frac{l \vdash e_l^1 : e_1' \qquad l \vdash e_l^2 : e_2'}{\vdash e_l^1\ op\ e_l^2 : e_1'\ op\ e_2'}$$

$$\textsc{Local:Negation}\quad \frac{l \vdash S_l : C}{l \vdash \neg S_l : \neg C} \qquad \textsc{Local:Conjunction}\quad \frac{l \vdash S_l^1 : C_1 \qquad l \vdash S_l^2 : C_2}{l \vdash S_l^1 \wedge S_l^2 : C_1 \wedge C_2} \qquad \textsc{Local:Disjunction}\quad \frac{l \vdash S_l^1 : C_1 \qquad l \vdash S_l^2 : C_2}{l \vdash S_l^1 \vee S_l^2 : C_1 \vee C_2} \qquad \textsc{Local:Implication}\quad \frac{l \vdash S_l^1 : C_1 \qquad l \vdash S_l^2 : C_2}{l \vdash S_l^1 \Rightarrow S_l^2 : C_1 \Rightarrow C_2}$$

$$\textsc{Local:Constant}\quad l \vdash n : n \qquad \textsc{Local:Variable}\quad l \vdash X : X \qquad \textsc{Local:Element Cardinality}\quad l \vdash \#\ell_l : N(l, \ell_l)$$

$$\textsc{Local:Sum}\quad \frac{l \vdash e_l^1 : e_1' \qquad l \vdash e_l^2 : e_2'}{l \vdash e_l^1 + e_l^2 : e_1' + e_2'} \qquad \textsc{Local:Subtraction}\quad \frac{l \vdash e_l^1 : e_1' \qquad l \vdash e_l^2 : e_2'}{l \vdash e_l^1 - e_l^2 : e_1' - e_2'} \qquad \textsc{Local:Multiplication}\quad \frac{l \vdash e_l : u}{l \vdash n \times e_l : n \times u}$$

$$\textsc{Local:Any Resources}\quad l \vdash \_ : \textbf{true} \qquad \textsc{Local:Resource Constraint}\quad \frac{l \vdash J_\phi : C}{l \vdash o\ op\ n; J_\phi : C(l).\phi(o)\ op\ n \wedge C} \qquad \textsc{Local:Repository Constraint}\quad l \vdash \bigvee_i r_i : \sum_i R(l, r_i) = 1$$

Table 6.7: Extended encoding: specification validity constraints

- Rule Local:Resource Constraint encodes constraints on resources available on a location using the variables.
- Finally, rule Local:Repository Constraint 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.

Full constraint problem

Definition 39 (Encoding extended architecture synthesis problem into constraints)
Given an extended architecture synthesis problem with input universe $\mathcal{U}$ and input specification S, the corresponding constraint problem $\mathrm{P} = \langle \mathcal{V}, \mathcal{D}, C \rangle$ *is defined as follows:*

- *the set of variables $\mathcal{V}$ and the function $\mathcal{D}$, attributing domains to variables, as in table 6.5,*
- *the constraints imposed on the variables $C = C_{\mathcal{U}} \wedge C_S$, where:*
  - *$C_{\mathcal{U}}$ is the constraint concerning the validity w.r.t. the universe $\mathcal{U}$, created using the translation provided in table 6.6,*
  - *and C_S is the constraint concerning the validity w.r.t. the specification S, created using the translation provided in table 6.7.*

Optimization

Now let us see how we can convert our optimization criteria, refined for use in the Zephyrus model, to a constraint form.

Compact We encode the *compact* optimization function as a multi-objective optimization based on three separate criteria: minimizing the deployment cost, minimizing the overall number of components and minimizing the number of packages:

1. We begin by encoding the first criterion, which minimizes the overall cost of the deployment, i.e. summary cost of all the locations which are not empty in the final configuration:

$$a_1 = \sum_{l \in C_l} \begin{cases} C.\texttt{cost}(l) & \text{if} \quad \exists_{t \in C_t} |C(l,t)| > 0 \\ 0 & \text{otherwise} \end{cases}$$

   This not too difficult thanks to the use of reified constraints:

$$opt_1 = \min \left(\sum_{l \in C_l} \left\| \sum_{t \in C_t} N(l,t) \geq 1 \right\| \times C.\texttt{cost}(l) \right)$$

   As we can see, mostly the conversion is direct and we only need to tweak slightly the formula which determines if a given location is used (i.e. is there is at least one component present on it) in order to convert the if-otherwise clause to the form of a reified constraint.

2. The second criterion, minimizing the total number of components in the configuration, is encoded exactly in the same way as our flat model *compact* optimization function:

$$a_2 = |C_c|$$

   becomes:

$$opt_2 = \min \left(\sum_{t \in \mathcal{U}_{dt}} N(t) \right)$$

3. Finally the third criterion, minimizing the total number of packages installed on all the locations in the configuration, is actually easier to express in the constraint optimization criterion form than using the elements of the Zephyrus model, as we have exactly the variables which count the overall quantity of each package, thus:

$$a_3 = \sum_{l \in C_l} \sum_{k \in C_k} \begin{cases} 1 & \text{if} \quad C(l,k) \\ 0 & \text{otherwise} \end{cases}$$

becomes simply:

$$opt_3 = \min\left(\sum_{k \in \mathcal{U}_{dk}} N(k) \right)$$

Now we can put these three single-objective optimization criteria together as a single multiple-objective lexicographic optimization criterion:

$$opt_{compact} = \text{lex}\left(\min\left(\sum_{l \in C_l} \left\| \sum_{t \in C_t} N(l,t) \geq 1 \right\| \times C.\texttt{cost}(l) \right) ; \min\left(\sum_{t \in \mathcal{U}_{dt}} N(t) \right) ; \min\left(\sum_{k \in \mathcal{U}_{dk}} N(k) \right) \right)$$

Conservative Encoding of the *conservative* optimization function is based on the same idea as in the flat model version: instead of checking what happened to each individual component, we treat them collectively. Only now, when counting the component differences, we take the component placement on locations into account. Also, we add to the count the differences in packages installed on each location. Thus the single-objective optimization function:

$$a_1 = \begin{cases} \sum_{c \in \mathcal{C}} \begin{cases} 1 & \text{if} \quad c \in dom(W_{init}) \wedge c \notin dom(W_{fin}) & \text{(case 1)} \\ 1 & \text{if} \quad c \notin dom(W_{init}) \wedge c \in dom(W_{fin}) & \text{(case 2)} \\ 1 & \text{if} \quad c \in dom(W_{init}) \wedge c \in dom(W_{fin}) \wedge W_{init}(c) \neq W_{fin}(c) & \text{(case 3)} \\ 0 & \textit{otherwise} & \text{(case 4)} \end{cases} \\ \quad + \\ \sum_{\substack{l \in \mathcal{L} \\ k \in \mathcal{K}}} \begin{cases} 1 & \text{if} \quad C_{init}(l,k) = C_{fin}(l,k) & \text{(case 5)} \\ 0 & \textit{otherwise} & \text{(case 6)} \end{cases} \end{cases}$$

becomes:

$$opt_1 = \min\left(\sum_{l \in C_l} \begin{cases} \sum\limits_{t \in \mathcal{U}_{dt}} | \, N(l,t) - |C(l,t)| \, | \\ \\ \sum\limits_{k \in \mathcal{U}_{dk}} | \, N(l,k) - \|C(l,k)\| \, | \end{cases} \right)$$

To construct the rest of the multi-objective *conservative* optimization criterion (i.e. after minimizing the differences we have to minimize the deployment cost and, finally, minimize the total number of packages) we reuse two of the single-objective optimization criteria encodings from the *compact* criterion, which gives us the following:

$$opt_{conservative} =$$

$$= \text{lex}\left(\min\left(\sum_{l \in C_l} \begin{cases} \sum\limits_{t \in \mathcal{U}_{dt}} | \, N(l,t) - |C(l,t)| \, | \\ \\ \sum\limits_{k \in \mathcal{U}_{dk}} | \, N(l,k) - \|C(l,k)\| \, | \end{cases} \right) ; \min\left(\sum_{l \in C_l} \left\| \sum_{t \in C_t} N(l,t) \geq 1 \right\| \times C.\texttt{cost}(l) \right) ; \min\left(\sum_{k \in \mathcal{U}_{dk}} N(k) \right) \right)$$

We should note that, as before, this is not a perfect encoding of the *conservative* optimization criterion defined on the Zephyrus model level, but it is as close as we can get in the collective representation of components.

6.6.3 Final configuration generation

Now we will discuss how our final configuration generation method (described in previous chapter, section 5.3.4) needs to be modified in order to work well with the extended architecture synthesis problem. Actually, it does not require to be changed too much, as the new elements added to the Zephyrus model do not cause any particularly difficult problems during the configuration generation.

Stating the problem

Input

- We are given the extended architecture synthesis problem inputs:
  - the input *universe* $\mathcal{U}$,
  - the input *initial configuration* C_{init},
  - the input *specification* S,

  and, optionally, the associated *optimization criteria*.
- We are also given the constraint problem solution σ, which is a correct and optimal (optionally) solution of the constraint problem P corresponding to our architecture problem instance inputs (as in definition 39 on page 140).

Output Our aim is to construct a final configuration $C_{fin} = \langle L_{fin}, W_{fin}, B_{fin} \rangle$, which is a correct (as in definition 36 on page 132) and possibly optimal (as in definition 37 on page 132) solution of the given extended architecture synthesis problem.

What has to be done? In order to do that we basically extend what we were doing before. In fact it is quite easy, because all the information is directly encoded in the solution σ and we do not need to do anything complicated:

1. The locations present in the final configuration are basically the same as the ones present in the initial one: they have the same name and they provide the same resources. The only thing that we need to do is to fit them with the right repository, using directly the repository variables $R(l, r)$, and a right set of packages, using directly the local package variables $N(l, k)$.
2. Then, instead of generating components globally using variables $N(t)$, we will generate them on the locations, using the variables $N(t, l)$.
3. Finally, the binding generation does not change at all, as bindings do not depend on the component placement.

As we can see, the component mapping W_{fin} and the set of bindings B_{fin} stay almost exactly the same (the only difference is that components now generated per location in order to keep their identity). The new part is generating correctly the locations with the right properties (i.e. the mapping L_{fin}), but as their identity in the final configuration stays exactly the same as in the initial one (i.e. we do not need to generate location names in a complicated way like we do for components), it is really simple.

Location generation

Let us start by generating the mapping L_{fin}, which defines our locations. This is simply done by taking the locations from the initial configuration and attributing them the right repositories and packages, as described in the solution. We define L_{fin} as follows:

- The locations keep their identity, hence the set of location names stays exactly the same as in the initial configuration:

$$dom\left(L_{fin}\right) = dom\left(L_{init}\right)$$

- For each location: the resources it provides are the same as before, while its repository and packages are defined by the solution of the constraint problem:

$$\bigwedge_{l \in dom(L_{init})} L_{fin}(l) = \langle\, L_{init}(l).\phi,\ r,\ \{k \mid \sigma(N(l,k)) = 1\}\,\rangle \quad \text{where} \quad \sigma(R(l,r)) = 1$$

We will refer to this construction as our *location generation method*.

Extended component generation

We directly adapt the flat model *component generation method* (see section 5.3.4) to our new context by performing it per machine. We construct the mapping W_{fin} in the following way:

- First we construct two sets, $\mathfrak{R}_t^l$ and $\mathfrak{G}_t^l$, for each component type t in the universe $\mathcal{U}$ and each location l in the final configuration C_{fin}:
  - Set $\mathfrak{R}_t^l$ contains the names of components of type t placed on location l which will be **reused** from the initial configuration C_{init}.
    It is the biggest subset of $C_{init}(l,t)$ whose cardinality is smaller or equal to $N(l,t)$. This means that if there are too many components of type t on the location l in the initial configuration then we remove some of them to get only $N(l,t)$ in the final configuration:

$$\bigwedge_{\substack{t \in \mathcal{U}_{dt} \\ l \in dom(L_{fin})}} \begin{cases} \mathfrak{R}_t^l \subseteq C_{init}(l,t) \\ \left|\mathfrak{R}_t^l\right| = \min\left(N(l,t),\ |C_{init}(l,t)|\right) \end{cases}$$

  - Set $\mathfrak{G}_t^l$ contains fresh names for components of type t placed on location l which will be **generated** in addition to the ones reused from the initial configuration C_{init}.
    Basically if there are less components of type t on the location l in the initial configuration than $N(l,t)$ then we keep all of them, and add new ones with the set $\mathfrak{G}_t^l$, making sure that we take *fresh* component names (so that they do not repeat):

$$\bigwedge_{\substack{t \in \mathcal{U}_{dt} \\ l \in dom(L_{fin})}} \begin{cases} \mathfrak{G}_t \subseteq C \\ \left|\mathfrak{G}_t^l \cup \mathfrak{R}_t^l\right| = N(l,t) \end{cases}$$

- In order to actually guarantee that all the newly generated component names are indeed fresh and unique, all the constructed sets $\mathfrak{G}_t^l$ should be *pairwise disjoint* and none of them should contain any *old* component names (i.e. present in the initial configuration):

$$\bigwedge_{\substack{t_1,t_2 \in \mathcal{U}_{dt} \\ l_1,l_2 \in dom(L_{fin})}} \Big((t_1 \neq t_2) \wedge (l_1 \neq l_2)\Big) \rightarrow \left(\mathfrak{G}_{t_1}^{l_1} \cap \mathfrak{G}_{t_2}^{l_2} = \emptyset\right)$$

$$C_{init_c} \cap \bigcup_{\substack{t \in \mathcal{U}_{dt} \\ l \in dom(L_{fin})}} \mathfrak{G}_t^l = \emptyset$$

- Then using these sets the construction of W_{fin} is quite direct:
  - The components in C_{fin} are those in all the sets $\mathfrak{R}_t^l$ and $\mathfrak{G}_t^l$:

$$dom\left(W_{fin}\right) = \bigcup_{\substack{t \in \mathcal{U}_{dt} \\ l \in dom(L_{fin})}} \left(\mathfrak{R}_t^l \cup \mathfrak{G}_t^l\right)$$

– And all components in $\mathfrak{R}_t^l$ or $\mathfrak{G}_t^l$ are of type t and placed on location l:

$$\bigwedge_{\substack{t \in \mathcal{U}_{dt} \\ l \in dom(L_{fin})}} \bigwedge_{c \in (\mathfrak{R}_t^l \cup \mathfrak{G}_t^l)} W_{fin}(c) = \langle l, t \rangle$$

We will refer to this construction as our *extended component generation method*.

Binding generation

Placement of components on location, as well as all the other extensions of the problem, do not influence bindings at all. Therefore, the *matching algorithm* introduced before (corresponding to the pseudo-code in figure 5.6) still works and does not need any modifications in order to properly generate bindings in the context of the extended architecture synthesis problem.

Conclusion

Now, as for the flat model, let us put the whole configuration generation in one definition and state precisely what we can call a final configuration generated using the constraint solving approach which corresponds to a given extended architecture problem instance and a given constraint problem solution:

Definition 40 (Generating the final configuration for the extended architecture synthesis problem)
In the context of a certain extended architecture synthesis problem instance and given a variable mapping σ, which is a correct solution of the constraint problem P *corresponding to this extended architecture problem instance (as in definition 39 on page 140), we define each configuration $C_{fin} = \langle L_{fin}, W_{fin}, B_{fin} \rangle$, such that:*

- *the mapping L_{fin} was build using our* location generation method*,*
- *the mapping W_{fin} was build using our* extended component generation method*,*
- *and the set B_{fin} was build using our* matching algorithm*,*

as a final configuration generated using the constraint solving approach that corresponds to this extended architecture problem instance and this solution σ.

6.7 Conclusion

In this chapter we have developed an extended variant of the flat Aeolus stateless model, which we call Zephyrus model. We used it to define the extended architecture synthesis problem, which incorporates the sub-problem of placing components on available locations in a correct and optimal way. Then we have adapted our previously established approach, based on constraint solving, in order to find solutions to instances of the extended architecture synthesis problem. All the elements of our extended model and approach have been formalized (and proven correct in [31]).

In the following chapter we will see how everything that was have introduced here has been used as formal foundation of a real-life tool called Zephyrus, which is capable of synthesising descriptions of distributed system deployments, based on our model and approach, by using an actual constraint solver.

CHAPTER 7

THE ZEPHYRUS TOOL

7.1 Motivation

The aim of this chapter is to introduce the Zephyrus tool and explain how it works. Zephyrus is one of the practical results of the Aeolus project. It is a tool that can be used to automate the design of distributed system configuration, based on the theoretical foundations set forth in the preceding chapters:

- the extended version of the stateless Aeolus model, which we call the Zephyrus model,
- the encoding of the *extended architecture synthesis problem* into constraints

both described in detail in chapter 6.

7.1.1 Scope

Let us first delimit more precisely what aspects of Zephyrus we will focus on in this chapter and what we will mostly skim through.

Architecture and design

We will describe in broad strokes how Zephyrus is built (i.e. give an outline of its architecture and design) and we will try to justify why it was build that way through discussing some important design decisions involved in its development.

However, it is not our goal here to give a complete in depth description of all the aspects of the Zephyrus architecture nor to document exhaustively its implementation (e.g. present all the data structures and algorithms used, quote parts of the code). In other words: this is not Zephyrus developer's manual.

Development process

When explaining how Zephyrus works we will sometimes reference its development history and mention how some parts of it looked before, why they were modified and how the solutions to certain problems were improved over time.

However, it is not our goal neither to tell the whole story of the Zephyrus evolution nor to document its development process. We will include this kind of facts in some situations when they are particularly relevant. [1]

[1] We should note that the context of the Zephyrus development has certainly influenced its shape in a significant way on many levels. Zephyrus is a prototype research tool and it was evolving pretty much alongside the associated formal model. The rapid

Bottom line

Our main aim in this chapter is to explain in a satisfying manner how did we implement the theory (as it was presented in the chapter 6) in practice with Zephyrus. And although this will require in several cases getting into the gritty details, we will try to avoid it wherever is not necessary.

7.1.2 Implementing the theory in practice

Let us recapitulate our situation before developing Zephyrus: we had a solid (i.e. proven correct, etc.) theoretical idea for how to perform the configuration synthesis, based on a strong formal foundation and the constraint programming approach. The main challenge of Zephyrus was to take the theory which we had developed and make it work in practice. We had to find the right concrete methods, tools and techniques to implement all the involved abstract concepts and build a real-life tool which would realize our approach. For example:

- Appropriate fragments of our formal model are describing the **input and output** of the (extended) architecture synthesis problem. We needed to have an actual concrete syntax for all these inputs and outputs, basically we needed to define the formats of the files which will be consumed and produced by the real tool.
- The **constraint solver** occupies quite a central place in our approach, but in all the previous chapters it was treated more or less like a black box capable of solving abstract constraint problems. Now we needed to actually choose a specific working constraint solver and be able to use it on real problem instances.

Thanks to the fact that our formal approach was proven correct, we know that when we realize it in practice in form of a tool, the results it produces are supposed to be correct too (of course in the limits of correctness of the implementation itself). Thus our main worry after simply *making it work* is the question of *making it work well enough*: its performance.

7.1.3 Conclusion

In this chapter we will try to explain how did we addresses these two essential issues concerning Zephyrus:

- **Efficacy**: how the theory was made to work in practice.

  In other words:

  How are the Zephyrus inputs processed to become the outputs?

- **Efficiency**: how the theory and practice were adapted to practical considerations, like especially working in a reasonable time frame.

  In other words:

  What did we do to make Zephyrus work faster?

We will do our best to interlace these two topics in a way that makes it all understandable.

incremental development process, necessity to constantly adapt the software to the changing model and to add new features quickly (in order to evaluate them in practice) explain many of the idiosyncrasies which we can encounter on the lower-level, when browsing the Zephyrus code.

7.2 General information and work schema

First we will introduce some basic information about Zephyrus and its workflow, which we are going to elaborate later.

7.2.1 Zephyrus building blocks

The code of Zephyrus itself is written entirely in the OCaml language [79]. It is compiled and built using the ocamlbuild system [38] and its OCaml dependencies are handled using the package manager OPAM [94]. Zephyrus is also using some other languages than OCaml and several external tools:

- JSON language [26] for raw input / output files
- ATD language [67] and Atdgen tool [68] for treating JSON input / output files
- our custom *Zephyrus specification language* for specification input files
- DOT language [40] for graphical output files
- MiniZinc and FlatZinc [90] languages for constraint problem files
- coinst tool [25] for handling some operations on package repositories and their meta-data
- our custom settings syntax for Zephyrus settings files

Zephyrus has been developed principally by me, the humble author of this document. Some parts of the code were authored by Michael Lienthardt. Zephyrus is in development from January 2013 and currently it amounts to around 10 thousands lines of code. It is distributed under version 3 of the *GNU General Public License*[51] and available in a git[114] repository hosted on GitHub[53] which can be found at the following address: `https://github.com/aeolus-project/zephyrus.git`

7.2.2 The basic workflow

The basic interface of Zephyrus with the world (i.e. the inputs it takes and the outputs it generates) corresponds exactly to the extended architecture synthesis problem as described in the previous chapter, one Zephyrus run corresponds to solving a single specific instance of this problem.

We will describe the Zephyrus workflow starting from a very high level perspective and then we will slowly refine this simplified vision with more details. In the most abstract way, we can see Zephyrus as a black box capable of solving the extended architecture synthesis problem: the inputs define the problem instance to solve, the output is a solution of this problem instance. In other words, when the user provides Zephyrus with a series of valid inputs (which correspond to the inputs of the theoretical problem), eventually Zephyrus responds either by producing an output (corresponding to the solution of the problem), or by answering that there is no solution (to this particular problem instance).

Let us recall how does the "interface" of the extended architecture synthesis problem look like:

- Inputs:
  - a universe,
  - an initial configuration,
  - a specification,
  - and optionally: optimization criteria.
- Output:
  - a final configuration (if it exists).

Our aim, when developing Zephyrus, was basically to make the real program inputs and outputs correspond to these theoretical ones.

We should note, that a *real* program is an entity that functions in a different context than the *abstract* theory behind it, and this clean view of Zephyrus input / output is somewhat simplified. The actual Zephyrus tool, in addition to the pure architecture synthesis problem's inputs that have been summarized here, takes also other options and parameters related in various ways to different practical aspects of its functioning. Similarly, in case if a correct final configuration does not exist, it outputs a concrete error message.

7.2.3 The input / output formats

Let us glimpse now rapidly on what are the real Zephyrus input and output formats, which correspond to various parts of our formal model.

Universe and configuration

Zephyrus universe and configuration (both initial and final) inputs and outputs are encoded in JSON syntax. JSON [26] is an ubiquitous lightweight data format used to transfer structured data between programs. It was chosen because it is simple, easy to parse, process and pretty-print, and at the same time relatively human-readable. An obvious alternative would have been to use XML [123], but it was deemed unnecessarily heavy (in this case) in comparison to JSON.

The structure of both universe and configuration JSON files corresponds as close as possible to our formal model. There are some differences caused by the fact that JSON-based data formats cannot be forced to exactly match the pure mathematical definitions in the model. This means that some JSON files may denote an incorrect universe or configuration (e.g. we can specify a multiset with repeating elements in a JSON file where there should be a normal set in the formal model). The possibility of encountering not only syntactically, but also semantically incorrect data must be taken into account when reading and treating the input.

In Zephyrus, the configuration files can be also alternatively output in a graphical form which tries to mimic our graphical syntax (used throughout this document) by producing a graph in the DOT [40] format.

Specification

The specification input file has a dedicated syntax designed to correspond as closely as possible to the abstract extended specification syntax that we have defined for the extended architecture synthesis problem (see table 6.1). We call this concrete syntax the *Zephyrus specification language*.

There is no point in making a separate EBNF[2] notation for the Zephyrus specification language, because the correspondence between this concrete syntax and the abstract syntax and is one-to-one (they are isomorph modulo parentheses). We should note that initially there existed an alternative JSON-based syntax for the Zephyrus specification files, but it was deemed completely unwieldy to use and too different from its theoretical counterpart.

Optimization criteria

In Zephyrus for specifying the optimization criteria we provide only a choice between several predefined possibilities:

- compact
- conservative
- spread

[2] Extended Backus–Naur Form [1]

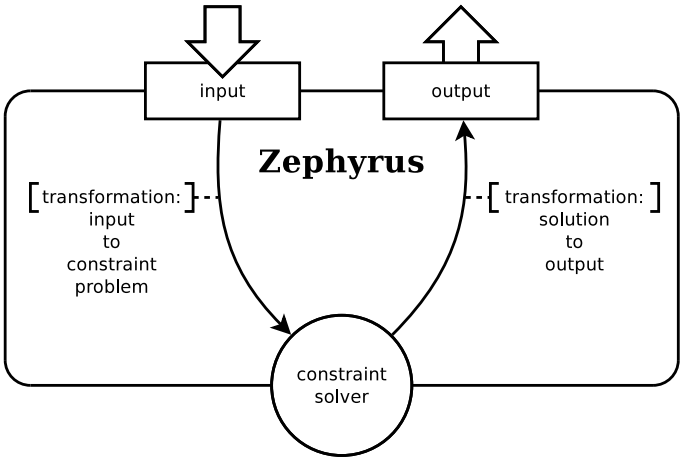


Figure 7.1: Simple Zephyrus workflow diagram.

We did not design any special format to manually define the optimization functions. It would be completely feasible: we could for example allow the user to specify directly an expression (using the variables already present in the constraint problem) which should be then minimized or maximized by the constraint solver. There are however good reasons why we did not implement it.

The first reason is that the choice of the optimization criteria influences greatly the shape of the problem that Zephyrus has to solve. It is extremely difficult to think about optimizing Zephyrus without focusing on a predefined optimization function which is at least basically sane (i.e. we know that it tries to minimize the deployment in some sense). If we let the user choose the optimized expression freely, we would be left mostly clueless about many aspects of the problem, for example which preprocessing heuristics are applicable in the given case or how the problem of this kind will be treated by the constraint solver, etc.

7.2.4 The principal arc of the workflow

The method that Zephyrus employs in practice in order to solve a given problem instance is a direct implementation of our theoretical constraint programming approach. Therefore the Zephyrus main workflow consists basically of these three essential steps (depicted in figure 7.1):

1. Translate the input into the form of a constraint problem.
2. Solve the generated problem using an external constraint solver.
3. Translate the found solution (if there is one) to the output form of a final configuration.

Implementation of both of the mentioned translations is directly based on their theoretical counterparts.

7.2.5 Elaborating the Zephyrus workflow

Of course this vision of what happens in Zephyrus is very simplified. In order to gain more insight on how Zephyrus proceeds, we will elaborate the description of the steps of this workflow. This basic schema will nevertheless serve us as a good frame of reference. We imagine the Zephyrus input, the constraint solver and the Zephyrus output as our three key points here. Each of the two big "transformations" forms an edge that links two points, and is in fact composed of many smaller steps.

"Way down"

Roughly this is the way from the raw Zephyrus input to the constraint solver:

1. Read the raw inputs from specified files and parse them: the universe (JSON-based format), the initial configuration (JSON-based format), the specification (Zephyrus custom specification syntax).
2. Check the input consistency and convert it to the Zephyrus internal model representation (i.e. strict OCaml data structures very closely resembling our formal model).
3. Pre-process the problem in order to improve its solving efficiency.
4. Convert the pre-processed problem to the form of an abstract constraint problem (corresponding closely to the one that we know (see table 5.1).
5. Translate the abstract constraint problem to a concrete form suitable for feeding to a specific constraint solver (e.g. MiniZinc constraint problem description language).
6. Launch the solver on the prepared input.

"Way back up"

And this is roughly the way back, from the solver output to the Zephyrus output:

1. Parse the raw solver solution.
2. Convert the solution to an internal representation of a constraint problem solution.
3. Generate the final configuration (in the Zephyrus internal model representation format) from the constraint problem solution.
4. Post-process the final configuration (i.e. restore all the pieces modified or trimmed during the pre-processing if needed).
5. Convert the final configuration to the required output forms (JSON, DOT graph, etc.) and print the outputs to specified files.

We will describe each of these steps much more in detail in the section 7.4.

7.2.6 Overview of the Zephyrus workflow

As we can see, the Zephyrus workflow is based on transforming the problem from one form to another on the way down (we start with several files containing the raw input, we end with a constraint problem ready to be fed to a solver) and retransforming the solution on the way back (we start with raw constraint problem solution generated by the constraint solver, we end with a final configuration in a printable form). Figure 7.2 gives a good image of this process.

The intuition here is that we have several forms of equivalent problem-solution pairs and we convert between them. We move one way when going "down" (when we transform the problem from one form to another) and then we go back on our trace when going "up" (when we transform the solution from one form to another). The whole workflow is centred on the constraint solver and more or less symmetric. We can, in some way, see the whole phase which happens before launching the solver (i.e. on the left) as pre-processing the inputs (in order to make them suitable to be fed constraint solver) and the whole phase after that (i.e. on the right) as post-processing the solution (given by the solver).

An important thing to notice here is that some information (which is lost when transforming the problem from one form to another) must be remembered on the "way down", because is later necessary to reconstruct the solution on the "way up". For example this is evidently true when we look on the constraint problem solution: if we forget its context, it becomes simply a mapping of some variables to values which is meaningless on itself. This is another good reason to represent the Zephyrus workflow as a shape similar to a ladder (or to the V-shaped software development model [49]) and not just as a single line of transformations.

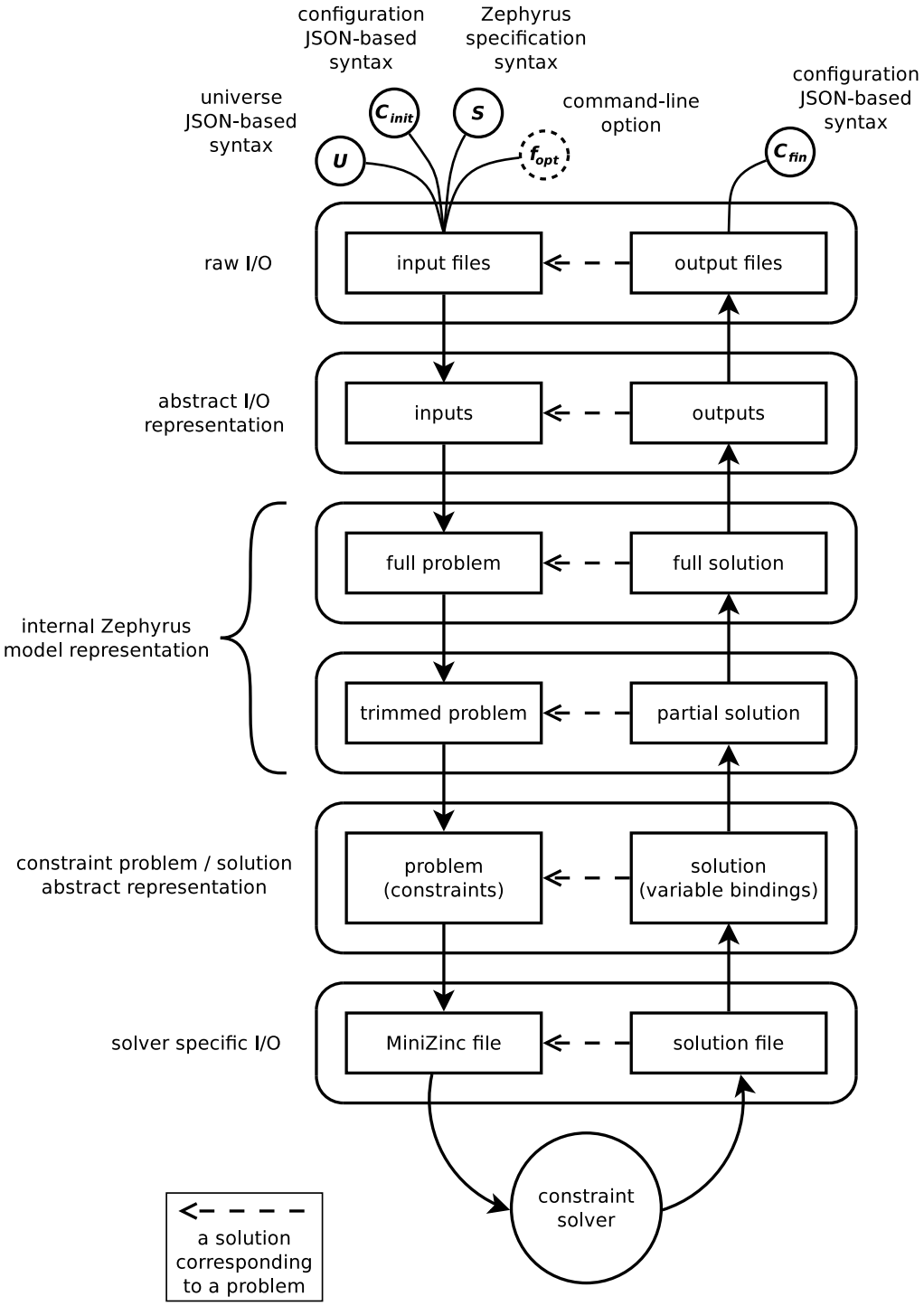


Figure 7.2: Zephyrus workflow depicted as problem-solution pairs and transformations between them.

7.3 Practical considerations

7.3.1 Introduction

We have established by now some basics concerning the general structure of the Zephyrus workflow and how do the key points of our theoretical approach correspond to our practical implementation. We have also caught a glimpse of how certain abstract concepts from the formal level are realized in Zephyrus on the actual operational level, like for example the idea of encoding the problem inputs in the JSON format. Now we want to look at the task of implementing our theoretical approach from a slightly different angle and consider how the approach which we have established needs to be adapted to practical considerations.

Until now we treated the development of Zephyrus as a very straightforward process of translating most faithfully possible our theoretical solution to an executable form. But in fact the reality is more complex than this simplified and unidirectional view, as the practical issues that we encounter influence our implementation and sometimes even force us to modify the theory behind it (we can see it as a sort of a software engineering feedback loop).

In our case the most important practical consideration is the performance. Up to now we have been acting as if the constraint solver sitting in the middle of our approach was a kind of a black box, which consumes finite domain constraint problems and outputs their (optimal) solutions. This may be a theoretically correct abstraction, but unfortunately it does not take into account one important factor: the time the solver takes to find these solution. Now if we want to build a tool working in the real world we are forced to actually consider this kind of mundane factor.

Therefore in this section we will focus on the question of efficiency: how do we make Zephyrus work fast enough to be usable in real situations. For now we will only establish some general principles concerning this issue. Later, after understanding more about all the steps of the Zephyrus workflow (described in detail in section 7.4) we will be able to proceed to discuss more deeply how does the application of our efficiency principles in practice works (in the section 7.5).

7.3.2 What is fast enough?

From the beginning a very important question concerning the implementation of our approach was:

- *Will it be fast enough in practice?*

Or more exactly:

- *How well will Zephyrus performances scale with growing problem sizes?*

Approximation of aim

First thing that we need to do if we want to start answering these questions is to define what time constraints the tool should respect in order to qualify as "fast enough". The definition of the acceptable time frame largely depends on how Zephyrus is to be used in practice. For example:

1. An execution time in the order of *several seconds* would make the tool accessible for an interactive use: a user could do as many repeated executions as she wants.

2. An execution time in the order of *several minutes* would mean that the tool needs some non-negligible time to compute the answer. However, we could still imagine the user relaunching it a few times with different parameters in order to explore the different solutions that it proposes in different cases.

3. An execution time in the order of *several hours* would mean that using the tool would require some planning ahead on the user side. It would however still be an acceptable time frame if we imagine that Zephyrus is used rarely, in situations which leave us with a comfortable margin of time, like designing a large and complex distributed system or planning major upgrades.

As we can see, even when considering these matters in a very simplified fashion, the required tool efficiency is not the same in different cases of use. Nevertheless, let us attempt to specify in this context what do we aim for when developing Zephyrus. Basically we would be happy if we could:

- treat *immediately* (i.e. few seconds) simple problems that need to be solved on-the-fly and deployed quickly,
- and treat in a *reasonable time* (i.e. few hours, maybe days) difficult complex problems corresponding to big industrial systems which need to be (re)designed very rarely.

This is of course a quite vague definition of aim, but gives us some direction.

Cloud feedback loops case

The paradigm behind all the scenarios considered here is that basically there is a user who asks (either directly or through some kind of interface) Zephyrus to solve a certain problem instance, then she waits for Zephyrus to synthesize a configuration, and finally after obtaining the solution she may either be done or she may have some further requests. However, we can also imagine another fashion of putting Zephyrus into work: a cloud-management feedback loop.

If Zephyrus got integrated into some kind of an automatic manager responsible of administrating a distributed system in the real-time (e.g. scaling it up or down depending on current load), then the execution time requirements would be potentially different to what we have discussed. On the one hand the real-time application would probably translate to stringier execution time requirements. On the other hand such a manager could potentially adapt the way it uses Zephyrus by taking into account the execution times for different kind of input, and it could thus, for example, plan ahead and prepare for particular situations in advance, hence compensating for the long execution times for particularly complex cases.

7.3.3 How to improve efficiency?

Either way, our main purpose here is stays clear: if we want to render Zephyrus more useful we should try to keep its execution times as low as possible (by working on its efficiency). Therefore, we should do what is usually done when optimizing software, i.e. attempt to identify the potential bottlenecks and proceed to remove them.

Identifying the bottlenecks

At this point we have seen the actual workflow of Zephyrus very briefly, thus it may be not clear where are the bottlenecks in it. The workflow itself will be detailed in section 7.4 and then we will see many examples of Zephyrus usage in chapter 8. And, as many benchmarks and tests prove (some of the most systematic ones are presented and discussed in section 8.2), in almost every case known to us the Zephyrus main bottleneck is, definitely and by a substantial margin, the constraint solving. The time required to do all the pre-processing (i.e. the "way down") and post-processing (i.e. the "way up") is mostly insignificant in comparison.[3]

These results are not very surprising, as no exact complexity calculations need to be performed in order to see very approximatively that the computational complexity of all the operations involved in these phases (mostly polynomial) pales when compared to the potential complexity of solving even moderately large constraint problems (finite domain constraint solving is in general NP-complete).

[3]There exists only one notable exception to this rule that is worth mentioning: distributed systems with a very simple structure, but a large number of components which need to be bound. In this case the constraint solving becomes almost trivial (few variables) and on the other hand the generation of bindings may take relatively more time. This kind of systems do not interest us very much, as they can be usually very easily designed by hand due to their simple structure, thus the gain from using Zephyrus is either way doubtful. In all the other cases that we have encountered it is definitely the constraint solving that takes almost all of the Zephyrus execution time.

Working on our principal bottleneck

Hence, our main effort should and will be concentrated on optimizing the constraint solving efficiency, which is our primary bottleneck. We can approach this matter from two sides:

1. By trying to influence the constraint solver itself and improving the process of solving the constraint problems generated by Zephyrus.

   In other words, we can try to **adapt the constraint solver to our problems**.

2. By working on the generated constraint problems before they are fed to the solver, in order to adjust them and make easier to solve.

   In other words, we can try to **adapt our problems to the constraint solver**.

The first of these ideas is unfortunately quite difficult to realize in a generic manner, because it highly depends on the specific constraint solver that we use. Moreover, trying to advance in this direction in a meaningful way requires some more advanced knowledge about the constraint solvers and lower-level details of their functioning. On the other hand there are still quite a few things that we can do in this context, which are conceptually simple enough. For example, the most basic way of influencing the behaviour of our constraint solver is simply replacing it with another constraint solver.

Anyhow, we will consider this idea more in detail in section 7.5 and for now we will rather treat the constraint solver as a black box and focus on the second approach: adapting the constraint problems generated by Zephyrus.

7.3.4 Working on the constraint problems

We will distinguish three main methods that we try to apply in Zephyrus in order to make our constraint problems easier to solve:

- **Simplify the problem**: reduce the problem's size and complexity without changing its solutions.
- **Divide and conquer**: separate the problem into orthogonal sub-problems, which can be solved faster than the original problem.
- **Make trade-offs**: modify the problem in a way that sacrifices some constraints or optimality of the solution, but leads to much faster execution.

Now let us take a closer look at these three ideas.

Simplify

The first idea is to attempt to simplify the problem without affecting its solutions.

Our premise here is the fact that the inputs for a given problem instance may often contain a lot of information not relevant to this particular instance's solution at all. This information will in many cases blow up the size and complexity of the generated constraint problem and make the solver's work more difficult, as it is not capable to discern easily between the relevant and irrelevant parts of the problem.

For example, our universe may define hundreds of different component types, corresponding to many available services, while in the current problem instance we might need only a few of them, which are easy to discern beforehand. We can thus remove the useless parts of the input (e.g. the irrelevant component types) and make the problem easier to solve for the solver without affecting the solution.

We can also try to make the generated constraint problem easier to solve by trying to express the same relations between variables using a different set of constraints (of course only if we are sure that they correspond to the same set of solutions). Not all ways of specifying the same problem result in an equally fast solving. We can adapt the constraint generation process in order to produce constraints that will be more efficiently solved by our constraint solver.

Divide and conquer

Another useful idea is to apply the *divide and conquer* principle to our constraint problems.

In order to make the problem *lighter* we may try to take some orthogonal "parts" of it, separate them from the main problem and solve them independently[4]. This way we take a part the problem's weight away from the constraint solver.

It is however important to understand, that by influencing our "main problem" in this way we will most probably influence also the corresponding solution that will be found by the constraint solver. Intuitively: as we take away a part of the problem (a certain sub-problem), we will also not obtain this part of the solution (a "sub-solution" which is a solution to this sub-problem). Hence this approach is acceptable only if we can at some later point regenerate all the missing parts of the solution and obtain the full solution (i.e. the same solution that we would have gotten if we had solved the full problem).

For example, an interesting application of the divide and conquer idea would be to separate *the packages problem* as a separate orthogonal sub-problem and solve it independently:

1. We imagine that we are able to replace the information about the packages in the problem input with equivalent information about direct incompatibilities between components (that means more or less: what components cannot be installed together on a single location).

2. The generated final configuration will be partially valid: all the components will be installed where they should be (the constraints normally imposed by packages will still be imposed with the same effect, albeit in a different manner), but there are no packages at all installed on any location, so the components are not properly implemented (as none of their implementing packages is installed).

3. But now we can simply regenerate the packages and add them to the final configuration. We know for sure that it is possible, because the components were placed correctly (i.e. in a way that there is always a way to co-install given components on the given machine). So on each machine we only have to find a valid (with respect to package dependencies, conflicts, etc.) set of packages that implements all the components present there and our final configuration will be completely correct.

4. If we do it well, in the end we obtain the same solution (i.e. the same as if we used our standard constraint problem), but we have taken the burden of treating the packages from the constraint solver and we have performed this task independently on the side.

Of course doing all that only makes sense if the gain in the constraint solver performance is significant enough to outweigh the cost of such a sequence of transformations and computations.

Trade-offs

The last of our three basic approaches to improving the generated problem's difficulty is to make some trade-offs: sacrifice obtaining the ideal solution in order to get a solution of a lesser quality, but faster. It may seem as quite a drastic measure, but on the other hand it is of course always better to have an imperfect solution than to have no solution.

Keeping that in mind, we have to admit that it is a not an easy approach to take. First, basically we really prefer to stay on the safe side, as we cannot afford to generate wrong solutions because then our tool will become largely useless (what is the point of using a formally proved theoretical approach if we do not profit from its advantages). Therefore, if we decide not to sacrifice the solution's correctness, the only other option is to sacrifice its optimality. And even if that is a tough decision, because the solution optimality is a strong point of our approach.

Nevertheless, even having made this resolution, it is not so easy to execute it properly. What we would like in fact, is to preserve at least some part of the search or optimality: maybe forgo asking for the optimal solution, but be able to ask for a solution of reasonable quality, e.g. "a solution in 5% of the best solutions". This is unfortunately really difficult to attain, as it would require us to find a way to judge somehow the level of optimality of a solution without a good frame of reference.

[4] We should note, that in fact we already did something similar to this idea on a fundamental level, by separating the system design and system deployment in the first place.

However, we should note that in some of our other efficiency-improving methods we may be obliged to do some occasional trade-offs. For example, if we look a little bit closer, we are in fact making a little trade-off in the "packages orthogonal sub-problem" case discussed just before: we ignore the possibility that in our formal model packages can consume resources (which may affect the installability and co-installability of components and packages on locations). When we employ this method it in practice we have thus two options: either we use it only when no resource consuming package is present in the repositories defined in our universe, or we must accept that our solutions may be potentially incorrect.

7.3.5 Conclusion

In this section we have discussed some principles concerning the the Zephyrus efficiency and previewed some ideas related to improving it. Many of the efficiency-related solutions basing on these ideas will be briefly foreshadowed when we pass more thoroughly through all the steps of the Zephyrus workflow (right in the next section 7.4). And after that (in section 7.5) we will see more in detail how do we apply those ideas to make the constraint problems generated by Zephyrus more manageable and easier to solve by the constraint solvers.

7.4 Zephyrus workflow details

7.4.1 Introduction

Let us examine now more in detail each step of the path that Zephyrus follows between reading the input and producing the output. In this section we will not enter too deeply into the questions of efficiency nor talk too much about all the methods employed to improve the Zephyrus performances (the parts of the Zephyrus workflow which are crucial to these topics will we be however clearly marked and examined more in detail in subsequent sections). Instead we will focus here on efficacy: what does Zephyrus do to attain the expected result.

In the meantime we will also attempt to establish a correspondence between what is described in each subsection and the actual OCaml modules implementing it in the Zephyrus code. A rough dependency diagram[5] of all the important OCaml modules in Zephyrus is available in figures 7.3 and 7.4.

7.4.2 Reading the input

The first thing that Zephyrus has to do in its workflow is to read the input provided by the user. The main inputs (defining the instance of the architecture synthesis problem that ought to be solved) have form of several files of several specific formats, which all have to be read and parsed.

The execution parameters

Of course Zephyrus needs to know where to look for all these input files. This information is passed to the program by the user through the **command line parameters** or the **settings files**. We will not discuss here in full detail how these two mechanisms work, we will only introduce the necessary basics. Also later on we will mention a lot of concrete examples of parameter passing when talking about the parts of the Zephyrus workflow that they influence.

Handling the command line parameters is implemented using the standard OCaml library module `Arg` [78], which is not extremely sophisticated, but gets the job done. An example of running Zephyrus with arguments that specify all its inputs looks like this:

```
./zephyrus -u    example/universe.json\
           -ic   example/initial-configuration.json\
           -spec example/specification.spec\
           -opt  compact
```

The meaning of these parameters is quite natural:

- the `-u` option makes Zephyrus search the input **universe** in the file `universe.json` in the directory `example`,
- the `-ic` option makes Zephyrus search the input **initial configuration** in the file `initial-configuration.json` in the directory `example`,
- the `-spec` option makes Zephyrus search the input **specification** in the file `specification.spec` in the directory `example`,
- and finally the `-opt compact` option makes Zephyrus use the *compact* **optimization function**.

[5]This diagram have been generated automatically as DOT [40] files using the ocamldoc tool [77] and then split in two parts and slightly tweaked by hand. The amount of possible manual customization is restricted in such automatically generated diagrams. In particular we can either choose to depict only the interface-level dependencies between modules, which basically gives us too little arrows between nodes to obtain a correct dependency graph structure (as many modules depend only on the implementation level); or we can depict all the implementation-level dependencies, which gives us a lot of arrows and a properly structured, but not very human-readable graph. We have opted for the second option, though we have used the tred tool [13] in order to perform a transitive reduction of dependencies and thus make the final graph a little clearer.

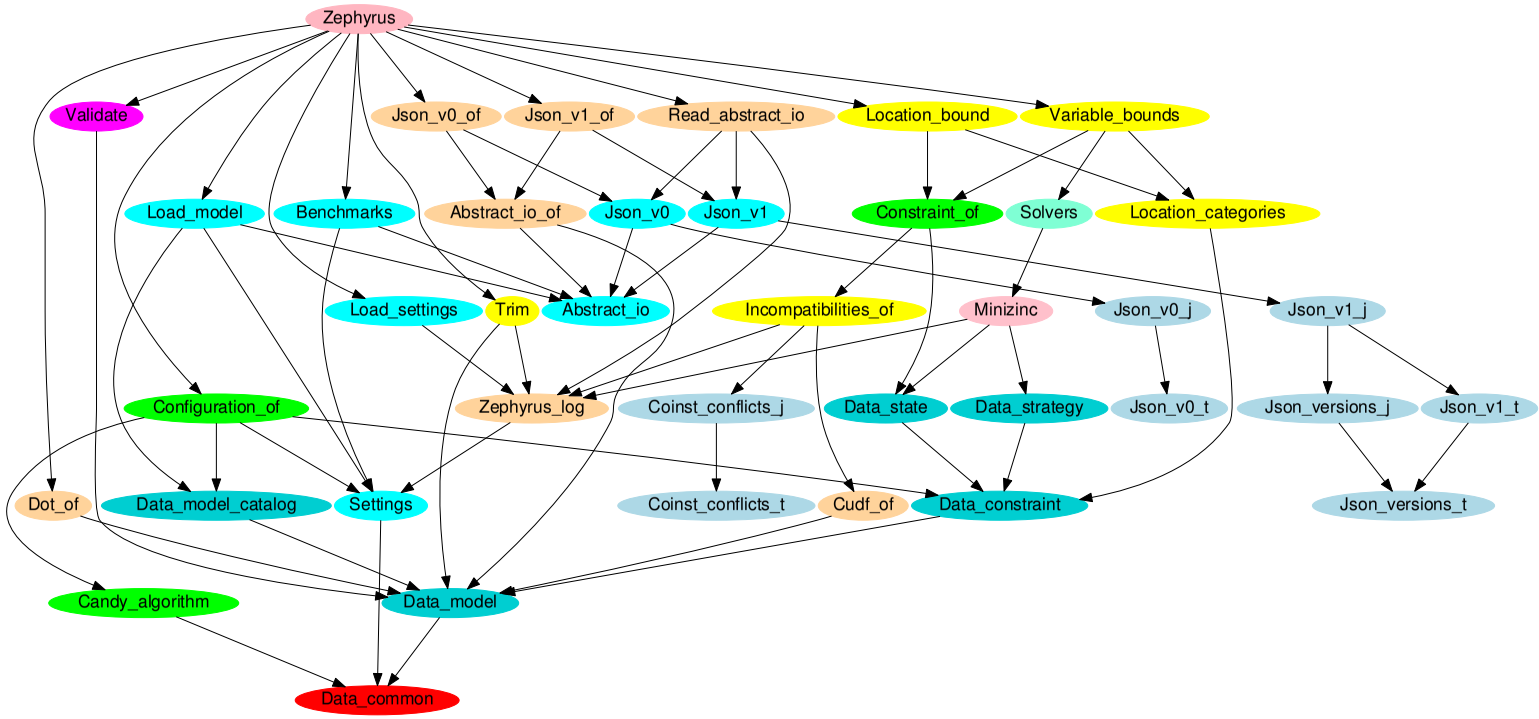


Figure 7.3: Diagram of the OCaml modules in Zephyrus. Slightly simplified: several helper modules were removed and all the DATA_COMMON_* modules are presented on the separate diagram in figure 7.4.

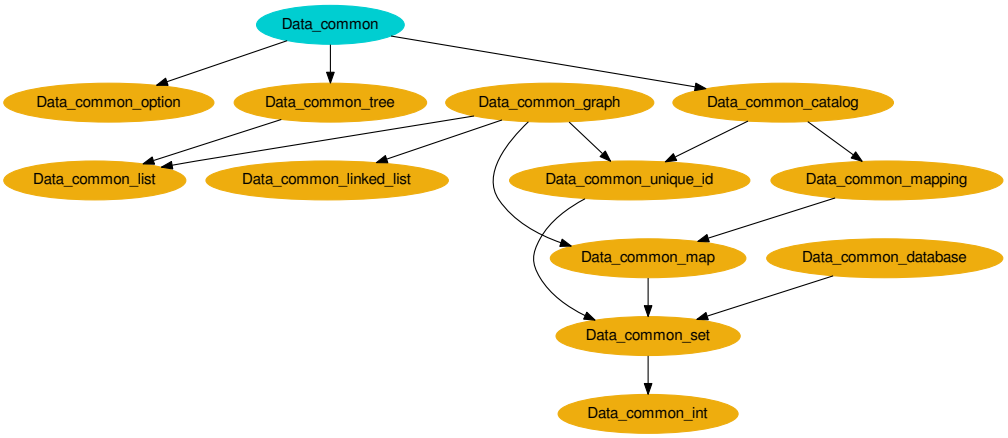


Figure 7.4: Diagram of the DATA_COMMON_* modules of Zephyrus, extending basic data structures and implementing several additional ones. We will not discuss them in this document, because these are mostly basic generic data structures used throughout Zephyrus code, but not particularly interesting in context of understanding how Zephyrus works on the higher level.

Another way of passing the parameters to Zephyrus is via *settings files*. The syntax of these files is rather simple, they basically contain pairs `key = value` (each in a separate line) defining the values of parameters. It allows not only for primitive values, like booleans, integers and strings, but also for lists and pairs.

Many parameters can be passed either using a command line argument or an equivalent setting file entry, depending on which method is more convenient to the user in a given case. A Zephyrus settings file equivalent to the presented command line call would look like this:

```
input-file-universe          = "example/universe.json"
input-file-configuration     = "example/initial-configuration.json"
input-file-specification     = "example/specification.spec"
input-optimization-function = compact
```

In order to make Zephyrus parse a specific settings file and take into account the parameters declared in it, we use a command line argument `-settings`. If we put the settings files contents that we have just presented in the file `all-inputs.settings` in the directory `example`, then with this call to Zephyrus we would pass it exactly the same parameters as with the previous one (which was relying only on command line parameters):

```
./zephyrus -settings example/all-inputs.settings
```

When the argument `-settings` is used multiple times, Zephyrus will read all the specified settings files, exactly in the provided order. Of course it may happen that the same parameter is declared multiple times in several settings files and / or through command line arguments. The strategy of treating such parameters is to systematically override the previously declared value by the new one, following strictly the order in which the settings files and command line arguments are provided. This mechanism permits the user to easily prepare modular chains of settings files. There also exists a special default settings file that is read always automatically at the beginning (thus all the parameters declared inside can be overridden later if necessary): `default.settings`.

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

(a) Zephyrus specification syntax in a file

$$
\begin{aligned}
& \#N(\texttt{wordpress-frontend}) = 1 \\
\wedge\ & \#(\_)\{\_ : \#N(\texttt{MySQL}) > 1\} = 0 \\
\wedge\ & \#(\_)\{\_ : \#N(\texttt{Wordpress}) > 1\} = 0
\end{aligned}
$$

(b) Zephyrus model formal specification notation

Figure 7.5: The same specification example in two forms.

The input files and their formats

As we have mentioned before, among the three main Zephyrus input files, two are in JSON format: the input universe and the initial configuration. The third one, the specification, is expressed using the custom *Zephyrus specification syntax*.

Custom specification syntax In order to handle the specification files, which have a completely custom format, developing an adapted lexer and parser was necessary. To this end we have initially used the standard OCaml lexer generator `ocamllex` [76] and the parser generator Menhir [50]. As we can see in figure 7.5, the concrete syntax used in the specification files is almost identical to the corresponding formal notation, so there is no point to elaborate this topic any more: it is indeed a very straightforward parsing exercise.

JSON-based syntax On the other hand, we had much more possibilities for treating the JSON files, as it is a very broadly used data format. Efficient JSON lexing and parsing libraries are available for virtually any existing programming language, including OCaml. Hence there was no need to write our own JSON-handling code from scratch.

However, we have found an even better alternative than using a JSON parser directly. Thanks to the language called ATD [67] we were able to define the structure of our JSON-based input files as OCaml data types and then use the associated tool Atdgen [68] to automatically generate adapted lexer, parser and pretty-printer modules.

The JSON syntax is quite simple [27] and consists mainly of several primitive data types, arrays (i.e. ordered lists of elements) and objects (in other words dictionaries, i.e. unordered sets of key / value pairs). The Zephyrus input JSON files (like almost all the JSON files designed for a specific program) are structured in a particular manner, by building higher-level patterns from the elements of the JSON language and, in certain sense, defining a custom syntax on top of the JSON syntax. We can see examples of a real Zephyrus JSON universe file (stripped of the package repositories) in listing 7.1 and an initial configuration file (empty initial configuration with 30 empty machines) in listing 7.2.

A standard JSON parser would permit us to read the JSON code element by element and to create a corresponding document tree in OCaml. But unfortunately we would have to verify by ourselves if the structure of this tree matches our data format and then convert the tree to a more usable form. Thanks to Atdgen we are able to automatically check if files given as input to Zephyrus are correct with respect to our JSON-based file syntax (separate one for the universe and the initial configuration files) and convert the structured JSON data directly to the form of native OCaml data structures.

The conversion between JSON and OCaml is specified using the ATD data definition language (by the way ATD stands for *Adjustable Type Definitions*) which bridges the gap between these two data represen-

Listing 7.1: A sample JSON universe file (repositories omitted for brevity).

```
{
  "version": 1,
  "component_types": [
    {
      "name": "DNS-Load-Balancer",
      "provide": { "@wordpress-frontend": "infinity",
                   "@dns": "infinity" },
      "require": { "@wordpress-backend": 7 },
      "conflict": [ "@dns" ],
      "consume": { "ram": 256 }
    },
    {
      "name": "HTTP-Load-Balancer",
      "provide": { "@wordpress-frontend": "infinity" },
      "require": { "@wordpress-backend": 3 },
      "consume": { "ram": 2048 }
    },
    {
      "name": "Wordpress",
      "provide": { "@wordpress-backend": 1 },
      "require": { "@mysql": 2 },
      "consume": { "ram": 1024 }
    },
    {
      "name": "MySQL",
      "provide": { "@mysql": 3 },
      "consume": { "ram": 512 }
    }
  ],
  "implementation": {
    "DNS-Load-Balancer":  [ [ "debian", "bind9" ] ],
    "HTTP-Load-Balancer": [ [ "debian", "varnish" ] ],
    "Wordpress":          [ [ "debian", "wordpress" ] ],
    "MySQL":              [ [ "debian", "mysql-server" ] ]
  },
  "repositories": [...]
}
```

Listing 7.2: A sample JSON initial configuration file.

```
{
  "version": 1,
  "locations": [
    {
      "name": "old-small-1",
      "provide_resources": { "ram": 1825 },
      "repository": "debian",
      "cost": 65
    },
    [...]
    {
      "name": "old-small-10",
      "provide_resources": { "ram": 1825 },
      "repository": "debian",
      "cost": 65
    },
    {
      "name": "old-medium-1",
      "provide_resources": { "ram": 4026 },
      "repository": "debian",
      "cost": 130
    },
    [...]
    {
      "name": "old-medium-10",
      "provide_resources": { "ram": 4026 },
      "repository": "debian",
      "cost": 130
    },
    {
      "name": "old-large-1",
      "provide_resources": { "ram": 8052 },
      "repository": "debian",
      "cost": 260
    },
    [...]
    {
      "name": "old-large-10",
      "provide_resources": { "ram": 8052 },
      "repository": "debian",
      "cost": 260
    }
  ]
}
```

tations (see listing 7.6).

Moreover, each ATD specification works both ways (i.e. defines the translation from JSON to OCaml and from OCaml to JSON), so as a bonus we gain a mechanism that helps us create output JSON files quite effortlessly.

Another advantage of this solution is its knowledge management value. Our ATD code serves in fact a double purpose: it not only allows Zephyrus to handle the JSON files (through Atdgen, which generates appropriate parsers and pretty-printers), but it also functions as a kind of a semi-strict specification for our JSON-based formats.

Importing external repositories

Zephyrus includes a feature that helps the user embed package meta-data from real package repositories into the input universe. This feature consists in fact of two sub-parts, one of which is not integrated into Zephyrus:

- First we use the coinst tool [25] to handle the actual package repository meta-data (it is able to treat both Debian or RPM meta-data formats) and prepare it for Zephyrus in form of so-called *external repository files*. For example, if the meta-data for the Debian "squeeze" release is in the file `Packages-squeeze-main-binary-amd64.gz` and we want to prepare a corresponding external repository file `debian-squeeze.json` (in the folder `repos`), we would run coinst in the following way:

  ```
  ./coinst -all -json -o repos/debian-squeeze.json\
           -deb Packages-squeeze-main-binary-amd64.gz
  ```

- Then, when we run Zephyrus, we can ask it to import these external repositories and include them in the input universe.

  The command line option used to include an external repository in the input universe is `-repo`, followed by the name we want to give to the imported repository and the path to the external repository file. If we apply it to our previous example of running Zephyrus it looks like this:

  ```
  ./zephyrus -u    example/universe.json\
             -repo debian repos/debian-squeeze.json\
             -repo ubuntu repos/ubuntu-14.04.json\
             -ic   example/initial-configuration.json\
             -spec example/specification.spec\
             -opt  compact
  ```

  Alternatively we can include the following declaration in one the parsed settings file for an identical effect:

  ```
  input-file-repositories = [
    ("debian", "repos/debian-squeeze.json");
    ("ubuntu", "repos/ubuntu-14.04.json")
  ]
  ```

The coinst tool does not simply read the meta-data and convert it to a different format. It also abstracts packages into a set of equivalence classes, which makes the repository several orders of magnitudes smaller in size, and yet does not lose any information about package incompatibilities. We will explain the purpose of this conversion and its repercussions more deeply a little bit later.

The external repository files format produced by coinst is JSON-based and corresponds exactly to the sub-tree of the Zephyrus universe JSON format which describes package repositories. We do not even need to write another ATD file to handle it, because the universe files parser generated by Atdgen is modular enough: Zephyrus simply reuses the appropriate sub-parser to read such a JSON file and then it directly adds the obtained repositories, already converted to the OCaml data structure form, to our input universe.

```
{
  "name"     : "DNS-Load-Balancer",
  "provide"  : { "@wordpress-frontend": "infinity",
                 "@dns": "infinity" },
  "require"  : { "@wordpress-backend": 7 },
  "conflict" : [ "@dns" ],
  "consume"  : { "ram": 256 }
}
```

(a) The universe JSON file fragment: a single component type represented in JSON. It is an example of JSON respecting the format defined by the ATD code below.

```
type component_type = {
  name : component_type_name;
  ~provide  <ocaml default="[]"> : (port_name * provide_arity) list <json repr="object">;
  ~require  <ocaml default="[]"> : (port_name * require_arity) list <json repr="object">;
  ~conflict <ocaml default="[]"> : port_name list;
  ~consume  <ocaml default="[]"> : (resource_name * resource_consumption) list <json repr="object">
} <ocaml field_prefix="component_type_">
```

(b) The ATD file fragment: this data type declaration defines the format of the JSON above and its correspondence with the OCaml types below.

```
type component_type = {
  component_type_name     : component_type_name;
  component_type_provide  : (port_name * provide_arity) list;
  component_type_require  : (port_name * require_arity) list;
  component_type_conflict : port_name list;
  component_type_consume  : (resource_name * resource_consumption) list
}
```

(c) The OCaml interface file fragment generated by Atdgen from the above ATD code: the `component type` record data type declaration.

Figure 7.6: Three corresponding file fragments: JSON file with a component type, the component type definition in ATD and the generated OCaml data type for the component type. We can see how the names of the fields of the top-level JSON object (which represents the whole component type) match directly the fields declared in the ATD file which, modulo the prefix (added there to avoid name clashes), correspond in turn to the OCaml record fields. Another interesting part the mappings in the JSON file (values of the `provide`, `require`) and `consume` fields) are also represented as JSON objects, but correspond to association lists (i.e. dictionaries implemented as lists of pairs) on the OCaml side.

As we can see, using an external repository meta-data is always a two-step process which is not entirely integrated into Zephyrus: we need to explicitly run coinst to create an external repository file and only then we can import it into the input universe. We should recall, that the real package repositories are often very big and usually must be pretreated in order to be suitable for Zephyrus. And, as coinst may take its time when performing this operation, we do not want to repeat the preprocessing of the same meta-data every time we run Zephyrus. Thus we leave managing the raw repository meta-data out of Zephyrus and we make it the user's responsibility to prepare the external repository files.

OCaml modules:

- Execution parameters:
  - handling command-line arguments: Load_settings
  - handling settings files: Settings
- JSON-related OCaml code generated by ATD: all modules with "_t" or "_j" postfixes.

7.4.3 The abstract I/O internal format

We have arrived at the point where Zephyrus has already read and parsed all its basic inputs: the universe, the initial configuration and the specification. Also the decision about the optimization criteria was made, albeit through the parameter-passing mechanism, as it is just a simple choice among few alternatives, not a complex data structure.

Now the next step in the Zephyrus workflow is to convert all the input data into an intermediate form called the *abstract I/O representation*. It lies somewhere in between a raw JSON file and the internal Zephyrus model representation used in the core of Zephyrus (which we will talk about shortly). Basically, the abstract I/O representation tries to be as close to the model as possible while relying only on simple language constructions such as lists and records, which are not very strict nor efficient, but instead offer a lot of flexibility.

This intermediate form was introduced at some point in the Zephyrus evolution for several reasons. The main reason was that our JSON files formats were slightly changing in time. Because of the fast prototyping approach to Zephyrus development, the first file format which was implemented was not necessarily the best one possible, so at some point we wanted to improve it. At the same time we wanted to keep backward compatibility and be able to handle the previous version files (so we do not have to sacrifice time and convert all the existing JSON files).

As the internal Zephyrus model representation is in fact quite far from the raw input format (in the sense that they have a rather different structure and purpose, even if are basically meant to represent the same thing), we have decided to introduce this intermediate unified layer between the two. From one side, this solution let us keep a single point in Zephyrus where we handle the complicated process of checking the input coherence and validity (the initial structural check performed when parsing the input is helpful, but it does not suffice) and of converting it into the more strict and optimized internal Zephyrus representation form (which is not a completely trivial conversion). From the other side the existence of this flexible intermediate layer lets us easily manage multiple input (and output) formats.

Thanks to the abstract I/O representation we can almost effortlessly juggle between the different Zephyrus input and output formats. This is a very practical feature (which has nothing to do with the principal Zephyrus functionality), that permits the user to translate files between different versions of our JSON-based formats, convert a configuration in JSON form to the graph form, etc.

OCaml modules:

- Operations on the abstract I/O representation: Abstract_io, Read_abstract_io
- Converting the JSON-based syntax to and from the abstract I/O representation: Json_v0, Json_v1

7.4.4 The internal Zephyrus model representation

After translating our input data into the abstract I/O representation, the time comes to convert it into the actual internal Zephyrus model form. This internal representation form was designed to have an interface as close as possible to the theoretical model, automatically assure the data coherence and at the same time be easy and efficient to process (access and modify).

Before and during the conversion process, all incoherences in the input data are spotted and reported or corrected. The abstract I/O representation assures that the input data has a proper structure, but it leaves a lot of place for other inconsistencies. For example:

- in the abstract I/O representation sets are represented using lists, so they are actually not sets but multi-sets: they may potentially contain repeated elements;
- as in the abstract I/O representation the references between different elements of the model are made simply "by name", there can exist some broken references, e.g. a package may depend on another package which in fact is not present in the repository.

The internal representation is strict and based on data structures with the properties that match the corresponding parts of the Zephyrus model. We use actual OCaml sets to implement sets of elements, OCaml maps to implement mappings, etc. All the names are converted to unique numerical identifiers and all the references between objects are checked to verify if their targets exist. The whole representation is easy to work on (i.e. access and modify in the program) and the operations on it are reasonably fast. Notably, thanks to a flexible architecture based on a mix of OCaml modules and classes, we can easily improve the implementation on any particular operation on this representation if it becomes important in the context of the overall Zephyrus performance, or add a new operation if it is needed.

OCaml modules:

- Internal Zephyrus model representation definitions: DATA_MODEL
- Converting the abstract I/O form to the internal Zephyrus model representation form: LOAD_MODEL

7.4.5 Pre-process the model

When we already have the architecture synthesis problem in the internal Zephyrus representation form, we can imagine that we have finally passed to the Zephyrus core part. The problem is now well defined and represented as strictly as in our formal model itself.

The next step in the Zephyrus workflow is when an important part of the actions aiming to improve the Zephyrus overall performance and solving efficiency takes place. Therefore we will only discuss these actions here quite vaguely as in this section of the current chapter we concentrate on efficacy and not efficiency. We will present more details in section 7.5.

What we need to know at this point is that after obtaining the problem in the internal Zephyrus model representation form, Zephyrus does not encode it into a constraint problem directly. Instead it pre-processes it, applying a sequence of different techniques that are supposed to render it easier to solve. This whole pre-processing takes the form of analysing various parts of the given problem instance, taking some parts of it out, solving some sub-problems on the side, etc. We can see that in terms of Zephyrus doing its best to prepare the ground for the constraint solver.

Often there is some information which is lost during this process, either simply removed from the problem instance or transformed into a different form. This information is remembered for later if necessary, in order for Zephyrus to be able to put it back in the model in the phase of building the final configuration. For example, if Zephyrus modifies the structure of the universe package repositories during pre-processing, it also keeps the full information about the original repositories.

We should also mention that the user has some control over the model pre-processing performed by Zephyrus. There are several parameters that can be passed to Zephyrus in order to tweak this procedure,

mostly by switching its different steps on or off. For example, the location trimming step (location trimming is an important efficiency-improving technique which will be presented in section 7.5.4) can be controlled with the option `-use-all-locations {on|off}`. By default the location trimming is activated (the default value of this option is `off`), but we can deactivate the trimming by providing the parameter `-use-all-locations on` and thus forcing Zephyrus to always leave the list of available locations, as they are specified in the initial configuration, intact.

OCaml modules:

- Basic component type and package trimming: TRIM
- Location trimming: LOCATION_BOUND, LOCATION_CATEGORIES
- Lifting conflicts to the component level: INCOMPATIBILITIES_OF (using CUDF_OF, COINST_CONFLICTS_J and COINST_CONFLICTS_T for interaction with the coinst tool)

7.4.6 Translate to the abstract constraint problem form

When the architecture synthesis problem in the internal Zephyrus representation form has been finally brought to a desired shape, we can move on to the next step of our approach. The time has come to convert our problem instance to the form of a constraint problem.

Basics The Zephyrus internal abstract constraint representation is almost entirely equivalent to the theoretical one that we already know (i.e. its syntax is practically the same as the one presented in table 5.1, including the optimization criteria as in table 6.4). Thanks to that fact, the translation mechanism applied here is a direct implementation of the one that was formally defined in the previous chapter (see section 6.6.2). All the variables and constraints are exactly the same, even the optimization criteria are identical. The only difference is that what we have seen in form of pure mathematical formulas Zephyrus actually builds in an operational way: it is traversing one data structure (the internal representation of the extended architecture synthesis problem instance) and creating another data structure (representing the constraint problem).

Tweaks This is however what happens only in the simplest case. Unfortunately the constraint encoding of the architecture synthesis problem that we have established before is only one of many possible encodings, and it is certainly not the most desirable one in terms of solving efficiency (in fact it is probably quite naive). Therefore, in order to make Zephyrus faster, we try to improve this encoding by tweaking it in various points: reformulating certain constraints in a more efficient way, adding new intermediate variables to avoid recalculating same expressions twice, introducing redundant constraints to reduce the search space and to eliminate symmetries in the solution, computing and imposing better variable bounds, etc. We will give a more comprehensive list of these improvements and explain how they are supposed to work in section 7.5.

Difficulty The main difficulty here lies in the fact, that there are so many places where we can tweak the translation, that it becomes challenging to manage. Individually each of these little changes is obviously correct (i.e. it modifies the generated constraints, but does not influence the solution), but we have to make sure that it stays correct in every circumstance, especially in presence of different combinations of other little improvements.

Result Either way in the end we obtain a formulated abstract constraint problem which corresponds to the given architecture synthesis problem instance. Our abstract constraint problem can be potentially pretreated and improved even further now, this time purely on the constraint level (e.g. by eliminating the constraints which are always true, propagating bounds). However, we did not implement any such additional preprocessing in Zephyrus, as is already performed very well by state-of-the-art constraint solver.

OCaml modules:

- Abstract constraint representation definitions: DATA_CONSTRAINT
- Translation to the abstract constraint form: CONSTRAINT_OF

7.4.7 A concrete constraint problem form

Now we have arrived at the moment when in our formal approach we would imagine feeding the generated constraint problem to a constraint solver and getting a solution. Unfortunately, in the real context, where we have to use an actual existing constraint solver, there is still some work to do before we can run it. We need to translate our abstract constraint problem to a concrete form which can be understood by the solver that we want to use.

FaCiLe episode

In the first version of Zephyrus this phase was not necessary. Initially we were using a solver called FaCiLe [14], which has a form of an OCaml constraint solving library. In this early stage in Zephyrus development, our abstract constraint problem representation was exactly the same as the format used by FaCiLe, thus there was no need to perform any more conversions: we could run an appropriate solving function directly on the abstract constraint problem.

The main problem with FaCiLe was that it often had trouble finding the answer, even in quite simple cases. That is because FaCiLe is more of a small academic project designed as a proof of concept than a full-fledged state-of-the-art constraint solver. It is thus rather slow and inefficient in comparison to many other widely available free solvers. Using it in the beginnings of Zephyrus was however very advantageous from the quick prototyping point of view: as FaCiLe is an OCaml library with a simple interface, it was particularly easy to integrate into Zephyrus. It has also provided us with a very good frame for some parts of the Zephyrus implementation, like for example our abstract constraint problem representation, which was heavily inspired by FaCiLe. Nevertheless, due to performance considerations, a decision to replace it with a more powerful solver was inevitable.

MiniZinc

When searching for a better solver, we have stumbled upon MiniZinc: a *medium-level constraint modelling language* [90]. We have found that:

1. There are many modern powerful solvers which accept MiniZinc input or at least FlatZinc, the *low-level solver input language that is the target language for MiniZinc* [90].
2. As the MiniZinc language itself is, at the fundamental level, reasonably close to our abstract representation of constraints (see table 5.1), converting our constraints to the MiniZinc form should not be too difficult.

Having considered these advantages, we have quickly decided to go in this direction and adopt MiniZinc as our concrete constraint description language.

The translation from our abstract constraint problem representation to the form of MiniZinc files proved to be almost direct. Basically the subset of the MiniZinc language which is sufficient to encode the Zephyrus constraints is so structurally similar to the internal constraint representation used by Zephyrus, that the translator is practically a pretty printer. In fact the only minor issue that has to be handled during the translating process is adapting the variable names, as MiniZinc rules for characters used in names are more strict than those used in Zephyrus. Other than that, the output of the Zephyrus debug-printing module is actually very similar to the corresponding MiniZinc code. We can see an example of a little part of a MiniZinc file generated by Zephyrus in the listing 7.7.

```
Bindings require:
((2 * N(Wordpress)) <= (B(@mysql, MySQL, Wordpress)))
((3 * N(HTTP-proxy-load-balancer)) <= (B(@wordpress-backend, Wordpress, HTTP-proxy-load-balancer)))
((7 * N(DNS-load-balancer)) <= (B(@wordpress-backend, Wordpress, DNS-load-balancer)))

Bindings provide:
((3 * N(MySQL)) >= (B(@mysql, MySQL, Wordpress)))
((1 * N(Wordpress)) >= (B(@wordpress-backend, Wordpress, HTTP-proxy-load-balancer) + B(@wordpress-backend, Wordpress,
 DNS-load-balancer)))
((1 * N(DNS-load-balancer)) >= (0))
((1 * N(HTTP-proxy-load-balancer)) >= (0))
((1 * N(DNS-load-balancer)) >= (0))
```

(a) A few Zephyrus constraints printed using a simple printer module which was created for debugging purposes.

```
%%   Bindings require:
constraint ((2 * global_component_wordpress) <= (binding__mysql_mysql_wordpress));
constraint ((3 * global_component_http_proxy_load_balancer) <= (
 binding__wordpress_backend_wordpress_http_proxy_load_balancer));
constraint ((7 * global_component_dns_load_balancer) <= (binding__wordpress_backend_wordpress_dns_load_balancer));

%%   Bindings provide:
constraint ((3 * global_component_mysql) >= (binding__mysql_mysql_wordpress));
constraint ((1 * global_component_wordpress) >= (binding__wordpress_backend_wordpress_http_proxy_load_balancer +
 binding__wordpress_backend_wordpress_dns_load_balancer));
constraint ((1 * global_component_dns_load_balancer) >= (0));
constraint ((1 * global_component_http_proxy_load_balancer) >= (0));
constraint ((1 * global_component_dns_load_balancer) >= (0));
```

(b) A fragments of the MiniZinc file corresponding to the constraints presented above.

Figure 7.7: Comparison of a debug printout of the internal Zephyrus constraint representation with the corresponding constraints in MiniZinc.

Multi-objective optimization issue

Unfortunately, even though MiniZinc seems to have been a very good choice, we have encountered one important issue while using it and solvers based on it: the multi-objective optimization.

Premise Our most essential optimization criterion, the compact optimization function, is in fact a multi-objective one. When we use it, we want to lexicographically optimize three different criteria: first to minimize the number of locations used, then minimize the number of components present, and finally minimize the number of packages installed. MiniZinc does not really support this kind of complex optimization criteria out of the box (i.e. in its standard syntax) and in fact there are not many (MiniZinc-accepting or not) constraint solvers that support it neither.

First idea: restrict solver repertoire One way out of this this problem would be to restraint ourselves to the solvers that do support this kind of optimization functions. For example the Gecode solver [105] is capable of handling the multi-objective optimization. And we can still use MiniZinc to encode them, as it is quite an extensible language capable of accommodating solver-specific data like that. Although this is a theoretically feasible solution, we did not decide to implement it, simply because it would work with only one solver (and even if we find another solver which can do that, it will probably not be encoded in the same way, due to lack of standardized MiniZinc notation, so we would have to reimplement it).

Second idea: weighted sum Another solution that we have considered is abandoning the idea of a optimization of several criteria in real lexicographic order (i.e. where each another is strictly less important than the previous one) and replacing it with an approximate lexicographic order: a weighted sum of objective functions. If we give a weight significant enough to the first criterion and then reduce it by a whole order of magnitude for each following one, it would work as a lexicographic optimization in all but most extreme cases, i.e. when a certain optimization criterion value happens to be an order of magnitude bigger that the previous one. Although implementing this solution would also certainly be feasible (at least if the number of criteria is limited), it is not only intellectually unsatisfying, but also not really safe, unless we find a good way to detect and avoid these extreme cases.

Third idea: simulate with single-objective optimization The solution that we have finally chosen is based on simulating the multi-objective optimization by performing multiple single-objective optimizations. It may initially seem to be not the best option, as it has several disadvantages: it is relatively difficult to implement and not very time efficient (the total solving time may increase linearly with the number of criteria to optimize lexicographically). However, it has also two important advantages which have a lot of value for us: it is completely safe and generic.

Basically the idea is to run the solver multiple times, reiterating the solving process for each objective. Every time we update the constraint problem with the results of the previous optimization, so that each optimization is performed in an environment where all the previous criteria are optimal (i.e. constrained to their optimal values).

The main problem with this approach to the multi-objective optimization is the fact that some things may have to be executed multiple times over and over again: our solver will probably repeat quite a substantial part of his search every time around. We can attempt to mitigate this drawback by trying to pass as much information as possible from one run to another, but unfortunately it is not very easy to do properly.

Conclusion Either way, for the time being the third approach, i.e. implementing the multi-objective optimization through performing multiple single-objective optimizations, is the only solution that is used in Zephyrus.

Other encoding ideas

Although the finite domain constraint solvers were our designated target from the beginning of our constraint programming approach, we have never rally ceased searching for potential alternatives. Unfortunately, we were not able to arrive at any promising results with the "lower-level" solvers (i.e. those that treat less expressive types of constraints, like SAT solvers). Even though the constraints generated by Zephyrus seem not to be that complex (in fact they are almost linear if we do not count the single essential variable multiplication: the one corresponding to the binding unicity principle and assuring that we can create correct bindings between our components), we did not succeed to find a solver of a less powerful category (in the sense of constraint expressivity) than a full finite-domain constraint solver and still able to treat them.

One of the ideas that we began to pursue, but did not finish, was encoding our constraints in ZIMPL language [72] and using a quasi-linear programming solver to tackle it. We have found a theoretical way to perform such a translation (albeit with a condition to restrict the expressivity of the Zephyrus specification language a little bit), but the initial implementation attempts encountered some difficulties and eventually the idea was dropped. However, it could be potentially worth reinvestigating at some point in the future.

OCaml modules:

- Translating abstract constraints to MiniZinc: MiniZinc

7.4.8 Solve

After generating constraint problem in form of MiniZinc, the next step logical step is to hand it to our MiniZinc-compatible constraint solver of choice and wait for it to come up with a solution (or find that there is none). Although, in theory this is not a very complicated phase in the Zephyrus workflow, in practice there is quite a lot of technical details that need to be properly handled. The whole procedure consists in fact of multiple steps:

- first we write the generated MiniZinc code to a temporary file,
- then we launch the solver with all the right parameters (the exact command line arguments that should be used differ between the solvers),
- at the end an output file is generated by the solver, we must know its path in order to retrieve and parse the solution.

Conversion to FlatZinc Moreover in many cases we also need to include an additional intermediary step: the conversion from MiniZinc to FlatZinc. That is because most of the so-called MiniZinc-compatible solvers do not actually accept directly MiniZinc input and they require the problems to be already translated into the lower-level FlatZinc format.

This is an example of a snippet taken from the Zephyrus log, that shows a bash command which Zephyrus runs when launching the solver (the bash command is in fact a single unbroken line of text, we have only added some line breaks here to improve clarity):

```
Running a bash command at pid 11543:
/bin/bash -c
mzn2fzn --no-output-ozn -o /tmp/zephyrus-c46331.fzn /tmp/zephyrus-a96db7.mzn
 &&
flatzinc -o /tmp/zephyrus-a30080.sol /tmp/zephyrus-c46331.fzn
```

We can see, that the translation from MiniZinc to FlatZinc (using the `mzn2fzn` tool from the G12 project) is integrated into one single bash command together with launching the solver. This is what we do most of the time, as it simplifies handling the external command execution from the Zephyrus side. All the temporary file names are generated beforehand in a safe way, using appropriate standard OCaml library functions.

Solvers The solver used in this example is the *G12 FlatZinc interpreter* (its executable available in the official Debian repositories is called `flatzinc`). The user can choose between available solvers by specifying the `-solver` parameter provided to Zephyrus. Currently there are 5 solving methods that we can select:

- The option `-solver g12` corresponds to using the mentioned standard constraint solver from the G12 project [112].
- The option `-solver g12_cpx` corresponds to using the G12/CPX (Constraint Programming with eXplanations) solver, also from the G12 project.
- The option `-solver gecode` corresponds to using the Gecode constraint solver [105].
- The option `-solver custom`, supplemented necessarily with the additional `-custom-solver-command` argument, lets the user provide manually the command that should be run by Zephyrus to call the solver. We can also use the `-custom-fzn2mzn-command` argument to replace the MiniZinc-to-FlatZinc conversion command.

  For example if we wanted to have exactly the same effect as in the previous example, we could run Zephyrus with the parameters:

  ```
  ./zephyrus -solver custom\
           -custom-solver-command "flatzinc -o <OUT> <IN>"\
           -custom-mzn2fzn-command "mzn2fzn --no-output-ozn -o <OUT> <IN>"
  ```

  The *custom solver* option gives Zephyrus quite a lot of flexibility, as it allows to use any MiniZinc-based solver. Moreover, as the presented command line options have their settings files equivalents, we can easily configure Zephyrus to use a given custom solver by default (i.e. do not need to specify these long solver commands all the time on the command line).
- The option `-solver portfolio` corresponds to a special solving method based on the principle of launching multiple solvers in parallel. Zephyrus runs our three basic solvers (G12 standard solver, G12/CPX and Gecode) simultaneously on the same MiniZinc input and waits. When any of them returns an answer, Zephyrus kills the other ones and continues the workflow with the obtained solution.

  As all three solvers are solving exactly the same problem, each of them would eventually find an equally optimal answer (if there is one). But as constraint solving can be quite unpredictable in terms of execution speed, it is difficult to guess a priori which of them will solve a particular problem instance in the shortest time. Therefore instead of guessing, we simply try all the alternatives in parallel and let the best one (i.e. the one most adapted to a particular problem instance) win.

We will cover the available solvers in detail in section 7.5.6.

An important thing to mention here is that Zephyrus uses all of these constraint solvers as external tools. They are not bundled in any way with Zephyrus and they are run through executing shell commands. The user has to install manually each solver which she wants to use and make sure that it is available on the `$PATH` in order for Zephyrus to access it.

OCaml modules:

- Handling the external solvers input: Solvers and several helper modules not included on our diagram.

7.4.9 Read the solution found by solver

From this point on we start going back up our imagined Zephyrus workflow schema (as depicted on figure 7.2). We have translated the description of the problem from one form to another several times, trying to improve its properties on the way, and in the end we have fed it to a constraint solver in form of a MiniZinc file. Now we assume that the solver has found an answer: an optimal solution to the presented

problem. If instead the solver found that there is no solution for our problem, then there is nothing else to do and Zephyrus exits with an error. But if there is a solution, then we have to submit it to a reversed version of all the transformations that we have applied to the problem description.

This series of transformations begins with two basic steps that will bring us our solution back to the level of an abstract constraint problem.

Parse the solution file First we read and parse the solution file generated by the solver. Although we need to write a custom solver in order to handle that, it is a very easy operation, as the constraint problem solutions are basically lists of variable name / value pairs, so the corresponding syntax is usually not very complex. This is how a fragment of a typical solution file generated by a MiniZinc-compatible solver looks like (in this case it was the G12 standard solver):

```
[...]
global_component_dns_load_balancer = 0;
global_component_http_proxy_load_balancer = 1;
global_component_mysql = 2;
global_component_wordpress = 3;
[...]
```

Reverse variable name conversion After parsing the solution file, we obtain a mapping from the MiniZinc variable names to their values. Now the only thing left to do is to reverse the variable name conversion that we did when translating the abstract constraint problem to the MiniZinc format (see section 7.4.7).

Reversing this conversion is the first time where we must use the information retained on the "way down" when going "back up" in the Zephyrus workflow. As the translation from the Zephyrus abstract constraint problem variable names to the MiniZinc variable names is potentially lossy (the MiniZinc variable names are based on a more restricted alphabet), the only way to reverse it is to recall exactly which internal Zephyrus variable name was converted to which MiniZinc variable name and now directly apply that knowledge to perform the opposite translation. And this is precisely what happens in Zephyrus: when generating the MiniZinc we create a mapping (i.e. a data structure) from the internal Zephyrus variable names to the MiniZinc variable names and then we can use it to reverse the translation.

These are the internal Zephyrus variable name / value pairs (as printed by the debug-printer to the Zephyrus execution log) corresponding to the MiniZinc ones presented just before:

```
[...]
N(DNS-load-balancer)        = 0
N(HTTP-proxy-load-balancer) = 1
N(Wordpress)                = 3
N(MySQL)                    = 2
[...]
```

OCaml modules:

- Handling the external solvers output: Solvers and several helper modules not included on our diagram.

7.4.10 Generate the final configuration

Now we came back to the level of the abstract constraint problem, this time however we are on the solution side. The time has come to perform one of the essential steps of our approach: generate the final configuration from the constraint problem solution.

This step is quite a complex and difficult one, but luckily we know already everything about it. We are now again in our closed and controlled environment inside what we could call the Zephyrus core, where our OCaml data structures and algorithms match very closely the corresponding concepts from our theoretical approach. Therefore the configuration generation process in Zephyrus is a direct implementation of the

```
let matching_algorithm requirers initial_providers =
  try
    (*The result variable accumulates the bindings between providers and requirers.*)
    let result = ref Results.empty

    (*The providers variable holds the current information about available providers and their
     capacity (i.e. provide arity).*)
    and providers = ref initial_providers in

    (*We need to bind each requirer to a number of different providers that he requires.  We iterate
     over all the requirers, treating them one by one.*)
    Requirers.iter ( fun (requirer_name, require_arity) ->
      (*requirer_name is the name of the currently treated requirer component.*)
      (*require_arity is the number of bindings with different providers that the current requirer
       component requires.*)

      (*The providers_left variable holds all the providers that this requirer is not bound to yet.
       With each step it will become smaller, as our requirer will be bound to another provider.*)
      let providers_left = ref !providers in

      (*We repeat the process of binding our requirer to a different provider as many times as it is
       necessary.  We use the variable to_do to count bindings that still need to be created
       (i.e. require arity which still needs to be satisfied).*)
      let to_do = ref require_arity in
      while not (Require_arity.is_zero !to_do) do

        (*If there are no more providers unbound to our requirer left, we have lost.*)
        if Providers.is_empty !providers_left then raise Impossible

        else
          (*We take the provider which has the most ports left unbound (and which is not bound to
           our requirer yet).*)
          let (provider_name, provide_arity) = Providers.max_value !providers_left in

          (*If the best provider available has no more ports left unbound, we have lost.*)
          if Provide_arity.is_zero provide_arity then raise Impossible

          (*Else, we bind the requirer to the provider and adjust all the variables accordingly.*)
          else

            (*We add this requirer-provider binding to the results.*)
            result         := Results.add          requirer_name provider_name !result;

            (*This provider is now bound to the current requirer, so it becomes unavailable,
             because all the providers bound to a given requirer must be different.*)
            providers_left := Providers.remove     provider_name !providers_left;

            (*This provider has one less unbound provide arity left for binding.*)
            providers      := Providers.decrement  provider_name !providers;

            (*One less binding needs to be created (i.e. one less require arity needs to be
             satisfied).*)
            to_do          := Require_arity.decrement !to_do;
      done;

    ) requirers;

    (*The algorithm is over and, as we have bound all the ports of all the requirers, we have won!*)
    (Some !result)

  (*We have encountered an unsolvable situation somewhere during the algorithm (an exception
   Impossible was rised).  Thus, it is not possible to correctly satisfy the requirements of all the
   requirers with the available providers.*)
  with
  | Impossible -> None
```

Table 7.1: The matching algorithm (in scope of a single port) implemented in OCaml.

configuration generation as described in chapter 6, subsection 6.6.3: we prepare the final configuration locations, then we instantiate the components and finally we create the bindings.

The *matching algorithm*, responsible for establishing correct bindings between components, is definitely the most complicated part in this whole procedure. In Zephyrus it was implemented by translating almost directly the theoretically-proven pseudo-code to OCaml. Thanks to the use of very abstract data structures we can in the same time be confident of its correctness (because it is almost identical to the proven algorithm's pseudo-code) and assure its efficiency (because we can easily put efficient data structures underneath). The `matching_algorithm` function in OCaml can be seen in figure 7.1 and compared with the pseudo-code from figure 5.6.

There is one important source of difference between the final configuration generation process in theory and in practice: the model pre-processing. We will not investigate this too closely here, but it is evident that in certain cases some post-processing of the final configuration may be needed at this point in order to put back the parts trimmed away (e.g. during the location trimming), reverse the modifications, etc. We should remember nonetheless, that not all the changes have to be always reversed and we might even want to modify the final configuration even further, it all depends on the situation and on what the user wants. For example, it makes no sense to restore the 1000 locations which were trimmed in the pre-processing if they only symbolized an unlimited amount of available virtual machines in the cloud; furthermore, in this case we should probably even remove all the empty locations that may be present in the generated final configuration.

OCaml modules:

- Generating the final configuration: CONFIGURATION_OF
- The matching algorithm: CANDY_ALGORITHM (the module's name corresponds to the former way of referring to our binding generation algorithm and should be eventually refactored to MATCHING_ALGORITHM)

7.4.11 Validate

The last step before outputting the final configuration is to check if it is correct. More exactly, to verify carefully and thoroughly if the final configuration that we have obtained in the end is really a legitimate solution of the architecture synthesis problem that we were given in the beginning: if it is valid with respect to the input universe and the input specification (see the validity definitions in the section 6.5 of the previous chapter).

This validity test is an important sanity check for us and a way to improve our confidence in the Zephyrus correctness. Although we know that our approach has been theoretically proven, there is always a lot of things that may go wrong in the middle, especially when we are so dependent on a third-party components such as constraint solvers.

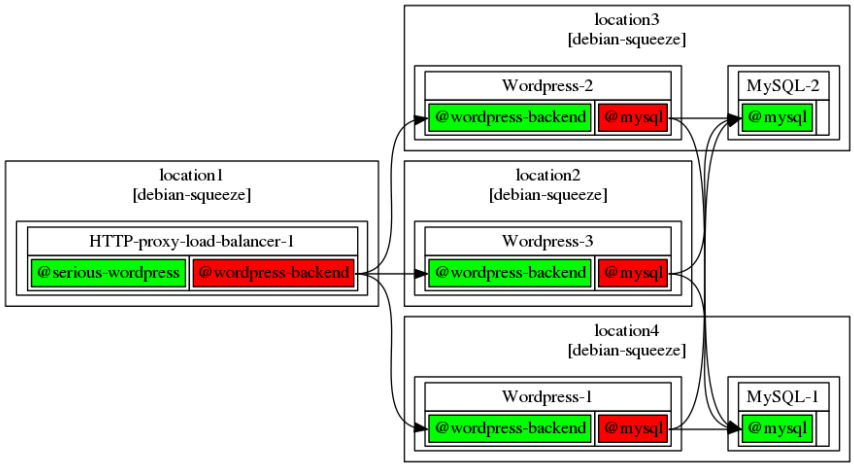
The test also helps us verify if there are no missing fragments in our model, that might have been trimmed on the "way down" and not restored on the "way up". We should always keep in mind, that all these model-related operations improving the Zephyrus efficiency are unfortunately not really based on solid theoretical foundations and thus it is easy to forget some crucial details when implementing them.

OCaml modules:

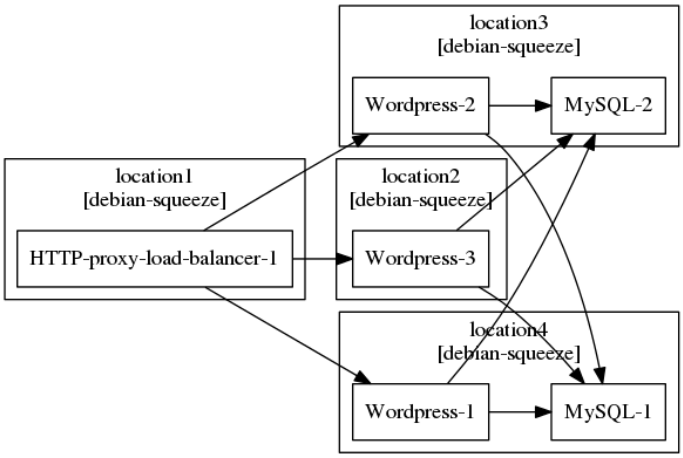
- Validating the final configuration: VALIDATE

7.4.12 Generate the output

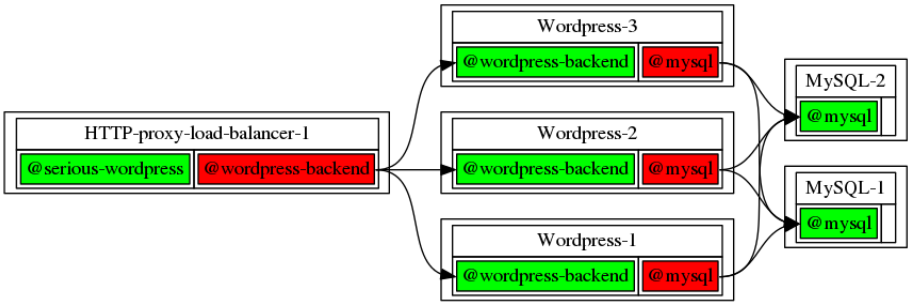
At the end of its workflow, Zephyrus produces the outputs. Actually the only kind of output that Zephyrus produces is the final configuration, as this is the only output of the architecture synthesis problem. It can write it out in two basic forms, each with some variations:



(a) Standard deployment diagram.



(b) Simplified deployment diagram.



(c) Components diagram.

Figure 7.8: Different diagram outputs of Zephyrus. (Green colour corresponds to provide ports, red to require ports.)

- A strict description of the final configuration in a JSON-based format.

  By default the latest version of our JSON-based configuration description format is used, but the user can specify a different version. The formats used for the final configurations are exactly the same as the ones used for the initial configurations.

  Zephyrus can easily handle the multiple JSON format versions thanks to the existence of the intermediate abstract I/O representation. First it converts the final configuration from the internal Zephyrus model format to the abstract I/O format (this conversion is always done in the same way). Then it is very easy to translate this representation in turn to a specific OCaml data structures generated by Atdgen for a given JSON format version and finally to print it to a file using the appropriate pretty-printer, also generated by Atdgen.

- A graphical representation of the final configuration as a graph in the DOT format.

  We could try to squeeze all the available information on a single picture, but it would quite unreadable, so instead Zephyrus offers a few different diagrams, each depicting a different aspect of the final configuration (we can see examples of these diagrams in the figure 7.8):

  – A standard *deployment diagram* tries to resemble our standard graphical syntax (as much as possible in the DOT format). On this kind of diagram we can see how the components are placed on the locations, what require and provide ports do they have and how exactly they are bound together.
  – A *simplified deployment diagram* offers us slightly less information. Instead of depicting faithfully the ports and showing each binding separately it only contains simple arrows to denote a dependency between two components (i.e. a binding going in a certain sense). Less information means however, that the diagram is cleaner.
  – A *components diagram* looks exactly like a deployment diagram, but it does not depict the component placement on locations.

  The DOT files are generated through our custom DOT printer module, straight from internal Zephyrus model representation of the final configuration.

The user can choose to produce as many output files as she wants in all the available formats. An output file can be specified by adding (any number of times) a `-out` options to the command line, like that:

```
./zephyrus -u    example/universe.json\
           -ic   example/initial-configuration.json\
           -spec example/specification.spec\
           -opt  compact\
           -out  json final-configuration.json\
           -out  graph-deployment deployment.dot
```

The equivalent settings file entry has a slightly different structure, as we declare all the outputs in one setting `results`, in form of a list of pairs:

```
[...]
results = [
           ("json",             "final-configuration.json");
           ("graph-deployment", "deployment.dot")
          ]
[...]
```

OCaml modules:

- Operations on the abstract I/O representation: Abstract_io, Abstract_io_of
- Operations on the JSON syntax version 0: Json_v0, Json_v0_of
- Operations on the JSON syntax version 1: Json_v1, Json_v1_of
- Generating the diagrams: Dot_of

7.4.13 Workflow variants

What we have presented here is the standard Zephyrus workflow. It is possible to enable different workflow variants which do not correspond to our theoretical approach, but may be useful in practice:

- **Stop-after-solving mode**

  If we only want to check if an (optimal) final configuration for given inputs exists, but we do not need to know how it looks like precisely, we can run Zephyrus in the *stop-after-solving* mode. In this mode Zephyrus exits midway through its workflow, as soon as the constraint solver answer is computed, without proceeding to generate the final configuration (i.e. basically it performs the "way down" part, but not the "way back up").

- **Validation mode**

  Zephyrus can be also run in the *validation* mode. In this case no solving is performed neither and in fact the normal workflow is completely disregarded.

  The initial configuration is simply matched with the input specification, using the information available in the input universe. All violations of the user specification are reported, with explanations of their origin.

  This modality is useful for designers who want to check whether a given configuration satisfies a request, and possibly understand why if it does not.

7.4.14 Conclusion

In this section we have presented the Zephyrus workflow, beginning with reading the inputs provided by the user and ending with writing the produced outputs. We have discussed in detail most of the parts of this process and also included some information about the actual Zephyrus usage (e.g. command-line options) related to them. This way now we can understand, at least on a basic level, both how Zephyrus *works* and how it is *used* in practice.

7.5 Improving the efficiency

In the previous section we have thoroughly examined each step of the Zephyrus workflow and hence gained a solid understanding of the essential part of its functioning. This way we have covered the topic of the Zephyrus efficacy: basically we know how does it manage to accomplish its aim and transform the inputs into the outputs.

Now let us focus on certain selected fragments of the Zephyrus workflow for the sake of investigating more in detail the techniques that we employ in order to improve its efficiency. We have already discussed the principles of our approach to the question of efficiency in Zephyrus (in section 7.3), now the time has come to see how these principles are applied in practice.

7.5.1 Reducing the input

The first efficiency-improving concept that we will pursue here is the idea of trimming and modifying the architecture synthesis problem instance before it is converted into constrains and fed to the solver.

The premise here is simple: in order to reduce the time that the constraint solver needs to solve the problems generated by Zephyrus (and thus increase the Zephyrus efficiency) we should try to reduce the size and complexity of these problems.

A little potential obstacle is the fact, that every time we alter a problem instance, we may possibly influence the set of its solutions, perhaps even unintentionally removing the originally optimal solution from this set. This may be dangerous, as removing the right solution is usually not something that we want. Thus we have to keep this issue in mind and tread lightly.

Safe input trimming

First we attempt to reduce the input's size and complexity by simply stripping the problem instance of everything which is not necessary to get the right solution. Our strategy here is quite straightforward:

1. First we establish which parts of the problem instance are superfluous (i.e. do not influence the solution).

   This often requires some more or less complex analysis of various parts of a given problem instance, like for example building a graph of component type dependencies and checking some of its properties.

2. Then we trim (i.e. remove) basically everything we can: all the parts of the problem instance which have been found superfluous.

   Often we need to remember some information about what was trimmed in order to be able to perform a symmetric operation on the problem's solution and bring the removed (or modified) parts back to their original form[6].

We call this operation, both the analysis and the removal phase together, *input trimming* (or *safe input trimming*, if we want to underline the fact that performing it does not influence the solution in any way).

Intrusive modifications

Sometimes we go further in our efforts to improve the efficiency and do things that are more intrusive. We modify the problem instance more than in the simple input trimming case (i.e. we remove more than only the superfluous parts, or we change the problem structure altogether) and we end up in a situation when the solver finds a different solution than it would find for the original problem.

Obviously, this kind of situation is potentially dangerous. Nevertheless, if we are capable of reconstructing (at some later point) the solution of the original problem from the solution of the modified one, then we are still in the clear.

[6]This is the same type of situation which we have already discussed on several occasions in section 7.4: in many parts of Zephyrus workflow something has to be remembered on the "way down" (when transforming the problem) and then recalled on the "way up" (to transform back the solution).

Applications

As we will soon see, there is quite a lot of easily noticeable opportunities to trim safely different parts of the Zephyrus input. Also we will present several possibilities of performing more intrusive problem modifications which seem interesting: they are potentially very advantageous for the solving efficiency reasons, yet mostly reversible at the solution level.

First we will see what is there to trim in the input universe, then we will move on to trimming the initial configuration.

7.5.2 Trimming the component types

When we think about our input universe as a repository of building blocks for our distributed system, one of the questions that comes to mind is: which of these blocks will be actually used in the final system?

Intuition

Our universe can contain hundreds of different component types representing all the different services that we might want to deploy in our distributed systems[7], while in practice only a small subset of these components will suffice for our needs in each given case.

For example, we can imagine that our universe contains component types corresponding to several web applications (many of which require a database connection) and also multiple database systems. If in a given case we do not want to deploy a web application at all, but our goal is only to deploy a database cluster, the information about the web application components is irrelevant for the task and we can safely trim it.

Practice

The constraint solver should not need to consider all the component types which are clearly out of scope for a given scenario. Even if including them in the problem does not change the final solution, they will still make the work harder for the solver, as it will have to treat all the variables and constraints corresponding to these component types. Therefore we can, and should, leave these components out of the constraint problem if we care about the solving efficiency.

In order to do it we perform an analysis of the component types available in the input universe in the context of what component types and ports appear in the input specification. We consider a graph of potential component type dependencies induced by the provide and require ports of each component. What we are looking for are all the component types mentioned in the specification or providing a port mentioned in the specification, plus all the component types that they transitively depend on; all the others have no reason to be present in the final configuration. Therefore it is enough to compute the subset of all the nodes reachable in this graph from the nodes indicated in the specification (i.e. either indicated directly or through one of their ports) to find the component types which are relevant to the given problem instance.

Let us see how application of this concept would look like in the example mentioned before (the one with web applications and databases). We would probably start with an input specification which asks for some particular database components or database-related ports. As we imagine traversing the graph of potential dependencies between component types, we would cover all the component types which might be potentially useful in the database cluster and omit all these which are only useful for web applications and not transitively required by any database component (like the core web application component types themselves).

Correctness

There is an important question which still needs to be answered: how do we know that trimming the component types in this fashion really does not influence the optimal solution?

[7] This of course depends on the way of modelling our distributed system. We can imagine approaches where we have one huge universe of component types and we use it for each deployment scenario, but we can also imagine approaches where we synthesise in a controlled way a precisely tailored smaller universe for each specific case.

Intuitively, if we assume the *compact* optimization criterion, the potential solutions that we lose in the trimming process are all clearly suboptimal. Any solution of the original problem instance which is not a solution of the trimmed one must contain at least one of the trimmed components. So if we simply remove all these components from this solution, we will obtain another correct solution. We know that it will be correct, because these components are intuitively ultimately useless: they may depend on each other, but nothing important depends on them. And the new solution obtained this way will be obviously strictly smaller and hence better than the initial one (in the context of the *compact* optimization function or any other optimization function which does not prefer adding useless components to an already correct configuration), thus revealing our initial untrimmed problem solution as suboptimal.

This is of course far from being a formal proof of correctness, but it seems to be obvious that at least in the context of our standard *compact* optimization function this technique would work well.

Gain

By performing component trimming we directly gain a reduced size and complexity of the generated constraint problem. Less component types means less variables and less constraints, thus less work for our constraint solver.

7.5.3 Trimming the packages

Next thing that we will inquire into is how to trim the package repositories present in our input universe. This is a very important matter (possibly much more than component trimming) because of two main reasons:

- The first reason is the potential **size** of those repositories: the number of packages to take into account.

  If we want our package repositories to correspond to the package repositories of real Linux distributions, we should expect tens of thousands of packages per repository. For example, the size of the Debian stable distribution has surpassed 40.000 packages already some time ago.

- The second reason is the **complexity** introduced by the relations between the packages.

  The complex manner in which packages can depend on each other and enter in conflicts with each other [3] makes managing the installed packages a very difficult task for the solver. In fact the package management on a single machine is quite a significant issue on its own, so when we multiply it by several machines and mix it into the already hard problem of the distributed configuration synthesis, all that becomes extremely difficult do deal with.

Techniques

There are three techniques that we will employ when trimming packages. One of them is similar to the component type trimming approach that we have just presented and the two others rely on the external package repository analysis capacities of the coinst tool.

1. The first method consists of analysing the structure of the input universe in the context of the input specification and removing all the packages that are irrelevant for the solution (this is the package equivalent to our component type trimming idea).

2. The second method, which involves using the coinst tool, reduces the size of our package repositories by an entire order of magnitude, but also alters their structure greatly. After the repository conversion performed by coinst, each of our model-level packages does not represent a single software package from the real world any more, but a whole group of packages, a coinstallability equivalence class.

3. The third method (which also requires the use of the coinst tool, albeit in a different fashion) is the most radical and intrusive one. It consists of completely removing the packages from the model and replacing them by a notion of direct incompatibilities between component types. Basically it means

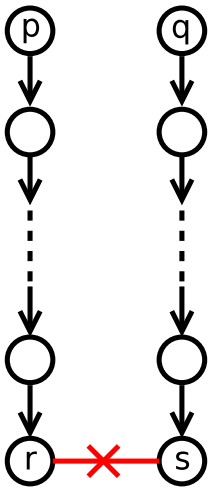


Figure 7.9: Packages p and q are in an indirect conflict, because their dependencies (somewhere far down the dependency chain) r and s are in a direct conflict.

that we are lifting conflicts to the component level, which permits us to get rid of packages and repositories altogether, as they are no longer necessary to assure the components' coinstallability.

If we want to embed an actual package repository (e.g. one corresponding to a certain Linux distribution) in our input universe, we are more or less obliged to use the first and the second of the proposed methods. If not, the constraint solver will be completely overwhelmed by the sheer number of variables and constraints associated with all those packages (multiplied by number of available locations).

Therefore in Zephyrus we apply both of them by default. In fact (as it was already mentioned in the previous section, subsection 7.4.2, when discussing importing the external repositories) either way we always need to prepare the meta-data of package repositories with coinst before feeding it to Zephyrus. In consequence in virtually all cases[8] Zephyrus will work on the already reduced repository (i.e. based on the coinstallability equivalence classes) instead of the original one.

Coinstallability equivalence classes by coinst

Let us look closer on the most important package repositories trimming method in our repertoire: restructuring a repository using package coinstallability equivalence classes found by the coinst tool. This method is very valuable to us, because not only it is capable of reducing the size of a package repository and the complexity of the relations between the packages inside it by an entire order of magnitude, but also we can be completely sure that it preserves the coinstallability relations (i.e. dependencies and conflicts) between all the involved packages, as correctness of this method has been proved formally [25].

Restructuring the repository The principle of the technique used by coinst is to analyse thoroughly the graph of relations between the packages in given a software repository (coinst can handle real package repository meta-data as input) and perform a wide range of transformations on it in order to clump together all the packages with similar coinstallability relations. In the end it produces a transformed repository where each pseudo-package represents a whole group of packages from the original repository: a single coinstallability equivalence class. "Coinstallability equivalence" means, that all the original repository packages, represented now by one pseudo-package, behave (and can be treated) exactly the same in the context of coinstallability. For example, there is almost always an equivalence class grouping all the "always installable" packages.

[8] Actually there is a way to force coinst to leave the original repository untouched and only convert it to the JSON format if we really want to.

Moreover, during the transformation process coinst is also taking care of reducing greatly the depth of the package relationships in the repository. In any normal package repository understanding the real relationship between any two packages may require the knowledge about a lot of other packages. Two particular packages can be for example in an indirect conflict with each other because some of the packages that they depend on are in conflict (as in the figure 7.9). This kind of indirect conflict relation is not easy to notice directly without analysing the whole repository structure. And coinst is performing this kind of analysis for us: it is rewriting this complex dependency / conflict graph to a simple two-level one, where each relationship becomes directly apparent. In the new (equivalent!) repository produced by coinst the relationship structure is much more shallow than in the original one and in order to fully understand the relationship between any two packages it is enough to check if there is a direct dependency or conflict between those two. No indirect relationships exist any more, they have all been "exposed" and made direct.

Advantages Advantages of constructing this new repository of pseudo-packages are enormous from the point of view of package management. For example, thanks to all the groundwork work done by coinst, checking if any set of packages is coinstallable on a given machine becomes very simple and quick (i.e. takes a constant time) as it does not require us to analyse the entire graph structure every single time.

From the Zephyrus point of view this pretreatment is also an immense help. The main reason of introducing repositories and packages into our model was, in the first place, determining if two services (implemented by some packages) can be coinstalled on a single machine. What coinst does to the repositories preserves for sure the coinstallability relationships between the packages. Therefore for us using the simplified repository produced by coinst seems to be a perfect solution: we reduce the number of packages and the complexity of relationships between them (both these factors influence directly number of variables and constraints that the constraint solver has to handle) and we do not affect their coinstallability in any way.

Disadvantages There is however in fact one piece of information that we lose if we decide to follow this path and transform our package repositories using coinst. Namely, we lose track of individual packages. Each real package becomes an anonymous member of an equivalence class. And, as the entities that we deal with now are the pseudo-packages representing these equivalence classes, we do not have any information about which exact package (or set of packages) from a particular class is really installed on each location. From the coinstallability point of view it does not matter: installing one package from a given equivalence class has exactly the same repercussions as installing all of them.

So what do we exactly miss because of this individual package anonymity? There are in fact three effects that this has on our approach, none of which is very severe:

1. First, the **resources consumed by packages** do not mean anything now.

   If we stick to the full representation of the package repositories, we could for example attribute to each package the disk space that it uses. And then we could use this information to check if what we want to put on a single machine with a limited disk space really fits there, or we could optimize the disk space usage, etc. Now, however, we are dealing with whole equivalence classes of packages and we do not know, on the model level, which subset of the real packages belonging to a given class will be really installed. Therefore attributing any resource-consumption related properties to these pseudo-packages is quite pointless.

   This is not a serious drawback, because in practice the disk footprint of a virtual machine is not a limiting factor for current deployments: indeed some recent research devoted to optimising the disk footprint of virtual machines [102] shows that current cloud providers offer virtual machines with a footprint over ten times greater than the minimal required, despite the availability of efficient tools that let compute this optimum in a matter of minutes, like `ami-thinner` [30].

2. Second, if we model real packages one-to-one, the final configurations that we produce contain all the necessary information about **exactly which software package has to be installed on which machine**. Right now, as the information that we obtain mentions only whole groups of packages, it

is effectively only usable as a way to assure the coinstallability of the involved components, because we do not know any more which subset of the packages from the group should be actually installed.

Fortunately this is not a big issue neither, as our principal purpose here was addressing the problem of component coinstallability. Gaining knowledge about the exact set of packages which should be installed on each machine was only an added benefit. And with this approach we still have the full guarantee that components which are co-located on a single machine by Zephyrus can be coinstalled together without introducing any package conflict. It means that for each machine there exists for sure a set of packages that fulfils all the package dependencies, avoids package conflicts and implements all the services placed on that machine. Therefore the sub-problem of finding this set can be left for later and solved separately from our main problem, for example by using a dedicated package management tool.

3. Third, there is one little question which arises when we perform the transformation from real-package-based repositories to their coinst-transformation-based equivalents: how do we **preserve the information about the implementation relation** between packages and component types? The implementation relation is crucial in our model, without it we have no link whatsoever between the services layer (represented by component types) and the underlying packages layer.

   There are two simple remedies to this problem:

   (a) We can inject all the real packages that implement any service to our new transformed repositories of pseudo-packages and make each of them depend on the pseudo-package representing its equivalence class. This way the connection obviously remains preserved, even if we will have a few more packages in our repositories than strictly necessary.

   (b) Or we can modify the implementation relation itself and move each component type's implementation from real packages to their corresponding equivalence classes. In other words we make each component type implementable by the pseudo-packages representing the equivalence classes which contain its former real implementing packages.

   The second option is slightly better, as it does not add unnecessary packages to the repositories. Unfortunately it is also more difficult to be integrated with coinst because of certain technical details, so for now only the first one was implemented in Zephyrus.

Conclusion The coinst-based approach to package trimming is very useful for us, as it basically helps a lot and causes little trouble. However, we should not forget about the fact that it interferes slightly with our model: we lose the possibility of expressing resources consumed by packages. Also it requires us to recreate the initial package repositories in the post-treatment of our solutions if we want them to be completely correct with respect to the input universe.

Basic package trimming

Another technique that we use in order to reduce the number of packages in our input universe is the basic trimming of superfluous packages. It is very similar in principle to the component type trimming presented before. Our strategy here is simple: in each repository in the input universe we identify all the packages whose existence is not influencing the solution of the problem instance, and we remove them.

How do we decide if a package is influencing the solution or not? Basically we verify if is has any effect on the installability or coinstallabilty of the available components. We use quite an operational way of selecting the packages that should stay in the repository:

1. First, we consider all the packages directly implementing any component type from our universe, i.e. those which are mentioned in its implementation relation. These packages, which we will call here *root packages*, must obviously stay in the repository no matter what, because otherwise we risk rendering some components unimplementable.

2. Then we consider the packages in the dependency cones (see [80] for details) of the root packages, i.e. all those that any root package transitively depends on. All the others are for sure useless in this particular problem instance, because they will never really need to be installed, thus they can be safely removed from the repository.

3. What interests us (inside all these dependency cones taken together) are only the packages potentially involved in any conflicts, plus all the packages that may depend on them or provide alternatives to them. In other words: we are interested only in the packages that may in some way create coinstallability issues between our components.

   Normally determining exactly which packages are that would be at least a little bit tricky. It would certainly require some deeper analysis of the graph of dependencies and conflicts between packages, especially because of all the indirect conflicts that can occur (see again the figure 7.9). Fortunately, if we restrain ourselves to the repositories pretreated by coinst, it becomes a much simpler task, because all the conflicts become direct.

It is more advantageous to perform this kind of package trimming after performing component trimming. By limiting the number of component types in the universe, we also limit the number of the implementation links between the component types and packages, which in turn leaves us with less of potentially useful packages in the repositories, hence more packages can be trimmed.

Direct incompatibilities

The most radical way of reducing the number of packages in the universe is to remove all the packages altogether and replace them with something entirely different: the information about direct incompatibilities between components. We can see it as basically lifting conflicts from the package level to the component level.

Motivation We have already abandoned the idea of representing each real package one-to-one with a package on the model level. If we hold on to the assumption, that the only purpose of modelling the packages is to assure the installability and coinstallability of our components, then expressing the information about installability and coinstallability directly and dropping completely the packages seems like the next logical step.

Potential issues There are two potential problems with this idea:

1. First, it is a little bit outside of the Zephyrus model as we have defined it.

   In the first package trimming method we were only changing *what the packages represent* in our model: instead of representing real software packages one-to-one, each model-level package was representing a whole group of real packages. This did not modify the Zephyrus model structure anyhow, and on the model level repositories and packages worked exactly in the same way as before.

   Now we are doing something much more intrusive: we are *replacing packages with a completely different concept*. The very structure of the model is modified and hence many formal definitions from chapter 6 are no longer valid.

   This is certainly not ideal. We would prefer to stick strictly to the model we have defined formally, as a large part of our whole approach was built on it.

2. Second, we how no idea a priori if expressing the component conflicts directly actually improves anything.

   Maybe encoding component coinstallability relations with packages is already the best that we can do in terms of solving efficiency? Of course it depends on how do we express the concept of "direct incompatibilities between components" in constraint form.

   We should underline that predicting if the solving time will be increased or reduced in this case is not a question of a simple comparison between the number of variables and constraints before and

after the trimming. Here the structure of these variables and constraints potentially changes greatly and the effect of such change on the solving time can be unpredictable.

Let us see, how we handle these two potential problems.

Outside the model Well, we start by simply ignoring the first issue. In fact in Zephyrus when applying this method we do not do it on the model level at all (i.e. on the Zephyrus internal model representation), but instead we postpone it until the constraint generation phase. This way we do not tamper with the model itself, only with the constraint problem. Of course this does not really make it better, but it pushes the "mess" away from the model pre-processing part of the Zephyrus workflow to the constraint generation part, so that the structure of the model itself is never altered.

Encoding the direct incompatibilities Let us then address the second issue:

- how to encode the direct component incompatibilities,
- and will this encoding be better in any way than representing the same information in form of packages.

We will begin by establishing some facts first. What we imagine, when we think about conflicts between components, are basically pairs of conflicting elements. However, what we want to express here exactly are not only conflicts between two given component types, but potentially between whole sets of component types (of an arbitrary size).

Two-way conflict What does it mean if two component types are not coinstallable?

For example, if we say "component type A is in conflict with component type B", it means that we *cannot* place together on any single location both a component of type A and a component of type B. Configuration containing such a component placement is invalid.

Conflict groups What does it mean then if a whole set of n component types is not coinstallable?

This idea may be in fact slightly counter-intuitive. If we say "components A, B and C are in conflict", it means that we cannot place together on any single location three components: one of type A, one of type B and one of type C. Any of the strict subsets of these does not create a conflict. For example, we can very well have a component of type A and component of type B together on one location and, as long as we do not also put a component of type C there, everything will be fine.

Therefore a conflict concerning a given set of elements does not imply a conflict of any of its subsets. Moreover, we should know that it is not possible express a n-element conflict using any combination of m-element conflicts, for $m < n$ [4].

Expressivity But why do we want to be able to express all that? Why do we not restrain ourselves to simple two-way conflicts? Because we need to mimic the expressivity that packages (with dependencies and conflicts as we have defined them) give us. And the n-element conflict groups exactly match their expressive power in the context of component conflicts.

Basic encoding So, basically our encoding of direct compatibilities in scope of a single repository is defined as a set of *incompatibilities*, where each *incompatibility* is a set of component types (a n-element conflict group). The incompatibilities have to be specified separately for each repository, because each repository can contain different packages with different dependency and conflict structure.

Broken component types Not all component types must be implemented by packages available in every repository. In repositories where there is no package implementing a given component type, this component type should be considered as broken and thus have his own one-element conflict group (this causes that it cannot be installed at all in a given repository)[9].

[9]Why? If we have packages in our model, then components are not installable by default on a random location: a package that implements them is necessary. However, if we do not have packages any more and we use direct component incompatibilities instead,

Formal definition So finally our encoding of the direct incompatibilities takes the form of a function from the set of repository names to a set of sets of component type names, denoting the component type incompatibilities in each repository.

Translation from packages to direct incompatibilities Now we know how we can encode the direct component incompatibility. However, we did not consider yet if translating the coinstallability information from a package repository form to a direct incompatibilities form is feasible nor how to do it in practice. In order to perform such translation we would need to extract (using the information about the repositories, packages and the implementation relation from our universe) all such sets of incompatible component types that correspond exactly to component type conflicts induced by each package repository.

This operation is indeed feasible, but it requires quite a complex analysis of the package relationships in our repositories. Fortunately, we do not have to worry about it ourselves, because it has been implemented for us as a feature in coinst [25] and in Zephyrus we take this solution as it is without worrying about the exact mechanism which makes it happen. We simply provide coinst with the necessary package information in CUDF [116] format and it gives us back all the component type incompatibility sets (in a simple JSON-based format).

Gain There is one issue which remains: the question if replacing the package repositories with the equivalent direct component incompatibility information helps. And the answer is: it depends.

It seems intuitively that if a package repository has a complex dependency and conflict structure which induces many local conflicts between the component types, then the number of the sets of incompatible components types can become impractically big, thus suppressing any advantage which may come from using this representation. On the other hand if there is little (or none) local conflicts between the component types, then the direct incompatibilities are potentially much more effective.

We did not study this problem too intensely and we did not arrive at any interesting theoretical conclusions. We did not succeed neither to perform any systematic and meaningful tests in practice. For the ones that we did perform the results between the two approaches seemed completely random. Thus further investigation into this question is required.

7.5.4 Trimming the locations

After investigating how we can remove superfluous information from the universe, the time has come to see if we can do a similar thing with the input configuration. And the only thing that we really can and want to trim in the initial configuration are the available locations.

Reasons The size of our constraint problem encoding is, in large part, proportional to the size of the set of available locations: the number of local variables $N(l, x)$ and repository variables $R(l, r)$ is obviously a multiplication of the number of all possible component types, ports, packages and repositories by the number of all the available locations. Hence reducing the number of locations that we have to take into account may be very helpful, especially because we often want to use machine parks with huge numbers of locations in order to simulate cloud environments.

Difficulties We will assume here that an optimization function aiming to minimize the number of used locations (e.g. *compact*) is in effect. Our aim is to only leave in the machine park those locations which may be necessary to obtain an optimal solution, or at least to remove as many unnecessary locations as possible.

Unfortunately it is not so easy to assure that we are actually not removing locations which are involved in an optimal solution without knowing such a solution beforehand. We have discovered that discerning the superfluous locations from the useful ones is in fact quite a difficult problem, which requires not

then components are obviously installable everywhere by default unless there is an incompatibility which forbids that. Therefore, in this situation we must explicitly make each component, which is not implemented by any package from a given repository, broken (i.e. self-incompatible) in that repository. Only then our incompatibility-based encoding of coinstallability will be equivalent to the package-based encoding.

only information about our available machine park, but also a full knowledge of the input universe and specification.

For example, a simple heuristic that we could design might be based on the ratio of resources provided per unit of cost for each location. We could determine this ratio for every location and then proceed to trim the locations beginning with removing the least cost-effective ones (i.e. those with the worst resources per cost ratio). But this basic heuristic fails miserably even on simple counter-examples: we might have some big machines which are very cost effective and some small machines which are less cost effective, but if the specification says that each of our n components must be placed on a different location, then it will still probably be cheaper to use n small machines than n big ones.

Implementation Our safe location trimming technique is based on the assumption, that in our typical machine parks there are usually many homogeneous locations with similar properties (which is quite reasonable for machine parks simulating clouds). We can group all the locations which are the same in one *location category* and then treat them collectively. As we group only indistinguishable locations in such way, it does not matter exactly which of them will we keep and which will we remove: only the number of locations kept in each category counts (in the context of a cloud environment intuitively it corresponds to instantiating a certain number of virtual machines).

We also profit from the fact that finding **any** solution of a given problem instance is usually a lot faster (by a whole order of magnitude!) than finding an **optimal** (e.g. `compact`) solution of the same problem. By combining this property with the concept of location categories, we can rapidly find how many of the locations in each category we actually need to be able to find any solution.

Basically, we look for the exact number by performing a binary search in each location category: in order to check a case "n locations of this category suffice" we run a non-optimized version of the given problem on trimmed machine park containing only n locations from the concerned category (and no other locations at all). This way we determine if it is possible to solve the problem using only locations of this kind and how many of them do we need at minimum. In the end, we trim our machine park to contain the found minimum of necessary locations from each category.

It is obvious that with this method we are completely sure that no useful location will be removed from the set available in the input initial configuration (assuming that we are using an optimization criterion minimizing the number of locations used, like *compact*). As it is possible to deploy everything correctly on only n locations of a given category (without employing locations from other categories), using more than that (e.g. $n + 1$) will surely led to an equally correct, but certainly more expensive, solution.

Conclusion This trimming method only works well when we have few different kinds of machines in our machine park. If every machine is different, we will simply not be able to remove any of them. Still, for very homogeneous machine parks it is quite effective.

7.5.5 Improving the generated constraint problem

After discussing various methods that we can use to reduce the size and complexity of Zephyrus inputs before even encoding them into constraints, let us now examine the encoding process itself, and the shape of the produced constraint problem, in search of potential ways to improve the overall Zephyrus efficiency.

From naive to efficient encoding

In the constraint solving terminology, the way how we represent a given problem using constraints is called *modelling* a problem (see e.g. [9, ch. 2]). Most problems can be modelled with constraints in many different ways, each of them equally correct and, in theory, equivalent. However, in practice two theoretically equivalent constraint models of the same problem may potentially result in a completely different solving difficulty, as one of them may be structured in a way more adapted to how the constraint solvers work (or how a particular constraint solver works) than the other.

The constraint modelling of the extended architecture synthesis problem which we have introduced through our encoding is very naive. Although of course it represents well the problem and it gives the right

solutions, it was not optimized at all to *help* the solver: we treat the constraint solver more or less like a black box and we do not take into account the way how it actually proceeds through the search space in order to find the correct (and optimal) values for the given set of variables. In our approach, the encoding from the Zephyrus model level to the constraint level is indeed quite direct, which is very advantageous from the theoretical point of view (e.g. we can understand it intuitively and prove relatively easily that it is correct), but not from the practical one (i.e. it might be unwieldy for the solver, and it has to do more work than necessary to solve it).

In order to improve the Zephyrus execution times, we can attempt to make the constraints that we generate more efficient for solving. As we can encode each problem instance in many different ways (which yield equivalent solutions), we can try to develop a new equivalent encoding solvable faster than our current one. In fact, this is a pretty standard concept in the field of practical constraint programming (see e.g. [9, ch. 9]): we often start with a fairly *naive* way of modelling a problem, but we work on it, refine it and search for alternatives until we arrive at an *efficient* modelling.

Use the problem structure information from the problem model level

The first natural technique which we can use to make the constraint model more efficient is to make good use of all the information that we possess on the problem model level. If we do not represent in form of constraints some regularities which exist in the problem, then the solver will not be able to profit from them directly in solving phase and will have to discover them on its own by laboriously exploring the search space.

Variable bounds For example, we can analyse the problem instance on the Zephyrus model level in order to infer better initial variable bounds on the component type variables, both global $N(t)$ and local $N(l, t)$, than the default ones which span from zero to infinity (for the "natural domain" variables). There are many ways to do it, e.g. through performing an analysis of the graph of the component type dependencies in the context of total resources available on all the machines.

Redundant constraints Another method of applying our higher-level knowledge of the problem is to add redundant constraints to the constraint model. When we restrict the search space in this way we help the solver, because (intuitively speaking) it may then discern and eliminate wrong solutions faster and thus traverse the search space more rapidly. Thus adding constraints usually makes the constraint problem *less* difficult than it was before, which might seem counter-intuitive at first.

An example of applying this principle in the Zephyrus encoding are what we will call here *Ralf's redundant constraints*. They come from a regularity discovered by Ralf Treinen, who noticed that for each component type t_r which requires a certain port p:

- either the number of instances of this component type given is equal zero,
- or the total number of instances of all the providers of port p (i.e. sum of number of instances of all the components t_p which provide it) must be greater or equal to the require arity of t_r on port p.

Formally, already expressed in constraints form:

$$\bigwedge_{p \in \mathcal{U}_{dp}} \bigwedge_{t_r \in \mathcal{U}\mathrm{R}(p)} \left((N(t) = 0) \vee \left(\mathcal{U}(t_r).\mathbf{R}(p) \leq \sum_{t_p \in \mathcal{U}\mathrm{P}(p)} N(t_p) \right) \right)$$

These constraints obviously do not change the set of correct solutions of the problem, but may help the solver in navigating the search space by letting it recognise faster if a given solution is not correct.

Restructure the constraint problem itself

We can also approach the constraint model optimization from another angle and attempt to restructure it a little more directly, without necessarily basing our actions in the regularities spotted in the model.

For example, if there is a common sub-expression repeating throughout several of our constraints, the solver will usually need to recompute it many many times during the whole solving process. If we create a new variable, with value fixed to the value of such an expression, and we replace all occurrences of the expression with this variable, we may greatly reduce this kind of recomputations, and thus thus speed-up the solving.

Symmetries

A very important factor deciding about the constraint model difficulty is the amount of the *symmetries* it contains. Basically a symmetry means that there are two (or more) correct and *different* constraint problem solutions which, from the problem model point of view, are completely *equivalent*. Such a situation is very disadvantageous, because basically (intuitively) the solver passes its time on finding the same solutions multiple times. Thus we try to eliminate symmetries whenever we can.

For example, one of the source of symmetries which we have to deal with in the extended architecture synthesis problem are similar machines:

1. If we have a certain number n of identical locations in our machine park,
2. and there is a certain, correct and optimal, final configuration C_{fin} which places some components on each of these machines, such that on location l_i a set of components C_i is placed,
3. the any permutation of placement of these component sets on these machines (i.e. any bijective function $f : \{l_1, l_2, \ldots, l_n\} \to \{C_1, C_2, \ldots, C_n\}$ defining such a placement) defines an equally correct and optimal final configuration.

In order to reduce the number of such symmetries, we include some additional constraints in our encodings. We basically establish an arbitrary order in scope of each group of identical locations and force them to respect an additional condition: every location in this order must contain as many or less components as the previous one. Formally, if we take n identical locations $l_1, l_2, \ldots, l_i$, we introduce following constraints:

$$\bigwedge_{2 \leq i \leq n} \left(\sum_{t \in \mathcal{U}_{dt}} N(l_{i-1}, t) \right) \leq \left(\sum_{t \in \mathcal{U}_{dt}} N(l_{i-1}, t) \right)$$

Although this method does not eliminate all the symmetries, it reduces their number while not discarding any *unique* (on the Zephyrus model level, not the constraint problem level) optimal solutions.

Solver behaviour in practice

Though all the presented ideas aiming to improve the way we model our problem are advantageous in theory, unfortunately our tests have shown that many of them do not have any considerable positive impact in practice.

Unpredictability Constraint solving is quite unpredictable in terms of estimating the solving time for different representations of the same problem. Choosing the most efficient among the available representations of a given problem is hard, even provided we have proper understanding of the alternatives (see [9, ch. 9]). Thus it is not easy to anticipate if a particular modification of the modelling will actually help or hurt in the case of constraint problems generated by Zephyrus.

What works In fact almost the only of the mentioned restructuring techniques which seems to really work is the elimination of symmetries caused by the component placement on identical machines. In other cases, for example *Ralf's redundant constraints*, the results were ambiguous. No systematic benchmarks were performed, as simply launching Zephyrus (with different constraint solvers) on several similar cases was enough to conclude that introducing these constraints seems to influence the solving time in a completely unpredictable fashion (i.e. sometimes reducing and sometimes increasing it, without any obvious regularity).

Reasons The suspected reason for this is that our solvers' default solving strategies are in fact not so simple as we imagined and they depend on the given problem's general "shape". Apparently modifying certain problem properties, like the number of constraints mentioning a given variable, may make the solver change completely its way of exploring the search space, sometimes for better and sometimes for worse. This makes introducing new constraints or variables quite a gamble.

Unfortunately attempts to replace our solvers' default strategy (tried on Gecode) with a more predictable manually prepared strategy failed miserably. All of our custom-made solving strategies proved to be simply far inferior in solving efficiency than the default one.

Conclusion Because of these discouraging results we have mostly abandoned the idea of improving the constraint model through working on the problem encoding. Instead we have concentrated our efforts on other approaches to ameliorate Zephyrus performances.

7.5.6 Choosing and adapting the solver

The last way of influencing Zephyrus performance which we will discuss is through directly manipulating the constraint solver itself, either by tweaking it or replacing completely.

Our solvers

In Zephyrus we have been using four different solvers:

- **FaCiLe** [14] was the first constraint solver which we have adopted. Basically it was a very good choice for early prototyping, but we have abandoned it quickly because of its sub-par performance (completely understandable by the way, as it was a proof-of-concept research project, never supposed to compete with state of the art constraint solvers). For further details see previous section, subsection 7.4.7.
- The standard finite domain constraint solver from the **G12** project [112] is the first of our two standard MiniZinc solvers. It is simply a very good constraint solver which was naturally integrated in Zephyrus because of practical reasons: it was automatically bundled with the whole G12 project suite that contained the `mzn2fzn` tool (MiniZinc to FlatZinc converter), an essential element in the Zephyrus workflow.
- On the other hand, our second standard MiniZinc solver, **Gecode** [105], was the best finite domain constraint solver around for a long time. It has won several MiniZinc competitions (gold medal in the MiniZinc Challenge 2008-2012 [89]) and the decision to use it in Zephyrus came from the fact that at this time in Zephyrus development Gecode was the fastest available free finite domain constraint solver on the market. It beats the G12 standard constraint solver almost in every case, though usually not by a large margin.
- Finally the **G12/CPX** solver (also from the G12 project [112]) is a somewhat a special case, as it is not based on typical search strategies, but on *Constraint Programming with eXplanations*. In most cases it outperforms both the G12 standard solver and Gecode by a whole order of magnitude. However, it is quite chaotic and unreliable and sometimes surprisingly it gets blocked or even exits with an error on quite simple cases.

Solving strategies

The MiniZinc language is expressive enough to let us specify a search strategy for the solver by hand. On the other hand, the solver is not obliged to obey these instructions. In fact among the solvers which we use only Gecode actually follows the search strategy provided in this way.

In MiniZinc we can influence two basic parameters which control how the solver advances through the search space:

- the choice of the next variable which should be treated,

- the method which should be used to split a given variable's domain.

Details are a little more complicated, but we do not need to get deeper into them at this level of abstraction.

Unfortunately our experiments with manually tweaking the strategy did not get us anywhere. We have tried encoding many different simple strategies which seemed sensible, but all of them resulted in terrible performances: we did not even manage to get close to the solving times assured by the default Gecode strategy.

The apparent conclusion is that the default Gecode solving strategy is simply too smart (and / or optimized on the lower level) that we could achieve anything better by such simplistic attempts to replace it. Thus we have abandoned this idea for the time being.

Portfolio approach

The portfolio approach is based on the principle of launching multiple solvers in parallel on the same constraint problem (for details see subsection 7.4.8). As different solvers may be more efficient in solving different types of problems, and in many situations it may be difficult to anticipate beforehand which solver will solve a particular case first, using them all simultaneously is a potentially advantageous strategy.

This approach is not only known to be effective in optimization problems [7], but also it seems very well suited to employ it in Zephyrus. In fact thanks to using a portfolio of three solvers:

- G12 standard solver,
- Gecode solver,
- and G12/CPX solver,

we can safely combine the "power" of the the G12/CPX solver (which works extremely well in many cases, but very badly in some) with the Gecode's and standard G12 solver's predictability (which work reasonably well in all cases). For now it is the most efficient approach that we have found in practice.

7.5.7 Conclusion

In this section we have proposed several techniques which could be used in Zephyrus in order to improve its efficiency. Due to limited time and resources these techniques have been actually implemented in Zephyrus in various degrees (see table 7.2) and not always tested and validated in a systematic fashion. Zephyrus is a prototype research tool, and the efforts to make it more efficient were only one of our many concerns during its development, performed in the spirit of rapid prototyping and exploration of possibilities. Further more systematic experimentation would be necessary in order to understand exactly how and when each of these efficiency-improving approaches is useful or not.

Technique		*Implemented? (Where?)*	*Remarks*
Input trimming			
Component type trimming		not implemented	unnecessary as in all current uses the component type universes are already minimal and there is nothing to trim
Package trimming	Package equivalence classes	implemented for imported repositories (by the coinst tool)	integrated in the default method of importing the external repositories
	Basic trimming	implemented (module TRIM)	always performed by default
	Direct incompatibilities	implemented as redundant constraints (module INCOMPATIBILITIES_OF)	direct incompatibility constraints are not used on their own: instead of replacing the constraints based on repositories and packages, they can be optionally generated and added as redundant constraints
Location trimming		implemented (modules LOCATION_BOUND, LOCATION_CATEGORIES)	optional, turned on by default
Improving the constraint problem			
Pre-computing variable bounds		not fully implemented (module VARIABLE_BOUNDS)	currently in wait for a complete implementation
Redundant constraints		two ideas implemented (module CONSTRAINT_OF)	*Ralf's redundant constraints* implemented and included optionally, direct incompatibilities used as redundant constraints
Restructuring the constraint problem		one idea implemented (modules DATA_CONSTRAINT, CONSTRAINT_OF)	added separate $U(l)$ boolean variables for used locations
Symmetries		one idea implemented (modules CONSTRAINT_OF, LOCATION_CATEGORIES)	ordering equivalent machines
Choosing and adapting the solver			
Solver choice		implemented (module SOLVERS)	G12 standard solver, Gecode solver, G12/CPX solver, custom MiniZinc solvers
Solving strategies		partially implemented (modules MINIZINC, DATA_STRATEGY)	implemented on the low-level (MiniZinc encoding), no mechanism to actually generate these strategies
Portfolio approach		implemented (module ENGINE_HELPER)	quite a low-level implementation

Table 7.2: Summary of efficiency-enhancing techniques mentioned in section 7.5 detailing to what degree they were actually implemented (and in which OCaml modules if it is relevant).

7.6 Conclusion

In this chapter we have presented the Zephyrus tool, which is one of the practical outcomes of the Aeolus project. We have explained how Zephyrus realizes our theoretical approach to solving the extended architecture synthesis problem (as defined in chapter 6) and we have discussed in details how it works on various levels. Now we will proceed to show some more tangible experimentation of its use.

CHAPTER 8

EXPERIMENTATION AND VALIDATION

8.1 Motivation

In this chapter we present the results of our experimental validation of the Zephyrus tool and of the Aeolus approach in general. We begin by discussing the outcome of a series of performance benchmarks which we have used to assess the Zephyrus' efficiency in solving instances of the architecture synthesis problem and to establish the tool's limits. After that, we introduce the other elements of the Aeolus tool-chain, Armonic and Metis, and show how Zephyrus fits in the big picture and contributes to performing software deployments in real-world distributed systems.

Finally, we report on a successful attempt to use Zephyrus in a corporate industrial environment, where it has been used to improve the efficiency of the continuous integration process and the deployment of test-environments.

We consider that this is clear evidence of the usefulness of Zephyrus and, by extension, of the value of the Aeolus model and the Aeolus approach to distributed system design.

8.2 Architecture synthesis efficiency

Although we have put a great deal of effort into finding ways to make Zephyrus work fast, solving an architecture synthesis problem instance still has a daunting complexity in theory. The bottleneck of our approach is, in almost all the cases, the constraint solving phase. Here we present the results of several benchmarks, which have served us to assess the efficiency of Zephyrus and guide our optimization efforts during its development process.

Our benchmarks are based on the WordPress farm use case, which has been already introduced and presented in detail in chapter 4, subsection 4.6.1. We have focused on two basic variations of this use case:

- in the **flat distributed WordPress** scenario we follow the description provided in chapter 4 with one important change: as we consider it in the context of the architecture synthesis problem, we convert it to stateless form;
- in the **fully distributed WordPress** scenario we refine the original use case (or rather its stateless form) in order to use it in the context of the extended architecture synthesis problem, which includes the component placement problem.

Both these versions are encoded in the Zephyrus model and parametrized, so that each scenario is scalable in several dimensions. This allows us to compare the Zephyrus tool performance on many variants of our scenarios, corresponding to different sets of parameter values, and changing execution conditions (e.g. trying several different constraint solvers). We investigate a large range of parameter values, ranging from realistic to quite extreme ones.

8.2.1 Flat distributed WordPress scenario

The first scenario which we consider is the *flat distributed WordPress* case. It is basically our standard WordPress farm use case, only converted from the stateful Aeolus model, as it was presented before, to the stateless Aeolus model form. Also, in order to use it with the Zephyrus tool, we must in fact encode it in the Zephyrus model, which means that we actually have to simulate the flat Aeolus stateless model with the Zephyrus model.

Conversion to the stateless form Making the original version of the use case stateless is fairly straightforward. Our stateful universe contains four component types:

- DNS load balancer
- HTTP load balancer
- WordPress
- MySQL

and each of them has exactly the same three states:

- `uninstalled`
- `installed`
- `running`

In theory, we could convert each stateful component type from the original universe to several stateless component types, i.e. one per state. For example, the stateful `WordPress` component type could give us three stateless component types: `WordPress-uninstalled`, `WordPress-installed` and `WordPress-running`. Similar thing would happen to all the other stateful component types in our use case (as all of them have the same three states).

However doing that seems unnecessary here, because most of the component types resulting from such a full conversion would be identical to each other from the model point of view. The reason is that all the ports of all our stateful components are active only in the `running` state and this is the only interesting state of each of the components. As they have no active ports in other states (i.e. `uninstalled` and `installed`), the stateless component types corresponding to such states simply have no ports at all. Thus, apart from their name, they are not only entirely interchangeable, but also their presence or absence does not change anything in the system.

Therefore we keep exactly one stateless component type per corresponding stateful component type: the one for the `running` state. The names of the component types stay the same. The resulting stateless universe is presented in figure 8.1.

Conversion to the Zephyrus model This conversion would be enough if we could use the stateless flat Aeolus model directly in our benchmarks. In practice though we cannot do that, because the Zephyrus tool works only with the Zephyrus model. Hence we need to make one more conversion in order to translate our flat universe to the form of a Zephyrus model universe.

This is a straightforward step, since Zephyrus model strictly extends the flat Aeolus model. The only important detail that needs slightly more sophisticated handling is related to the way the components are

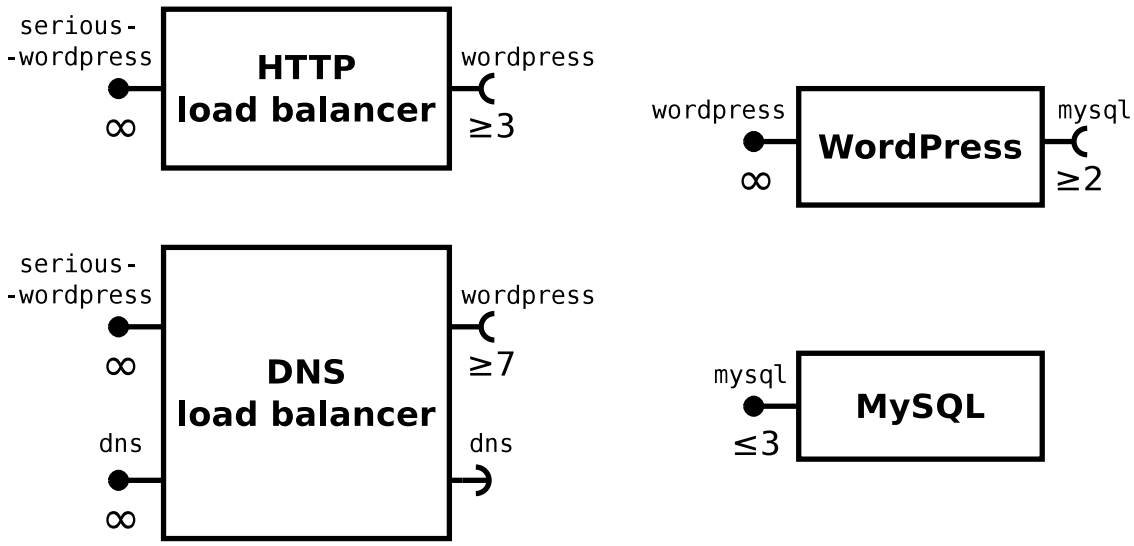


Figure 8.1: Conversion of the distributed WordPress use case: from stateful to stateless form.

implemented by the packages in the Zephyrus model. In fact if we ignore the repositories and packages altogether (i.e. do not include any in our universe), none of the instances of our component types are installable on any location, because there are no packages which could implement them. To avoid this problem we therefore add one repository to our universe, which is called the `Stub Repository`, with a single package inside, called the `Stub Package`. We also make this package implement all our component types.

Thus adapting the flat universe to a Zephyrus model universe for our use case is quite simple:

- The description of the component types does not change in fact, as they do not consume any resources.
- The set of repositories of our universe contains one repository: the `Stub Repository`. This repository contains in turn a single package: the `Stub Package`. And this package does not have any dependencies nor conflicts, neither does it consume any resources.
- The implementation relation is simple: each of our four component types can be implemented with one package: the `Stub Package`.

Parametrization As we have mentioned before, our aim here is not only to develop a single scenario suitable to base some benchmarks on it. We would rather want to have a a whole series of similar scenarios of different size and different level of complexity. This is why we parametrize our flat WordPress use case and make it scalable in several dimensions, so that we can easily generate various versions of it.

A WordPress farm is a distributed system that is inherently very scalable and we can easily represent this scalability in the Aeolus model (and of course Zephyrus model too). We simply define a whole family of universes where certain require and provide arities are parametric.

Our universe family is based on our WordPress distributed flat stateless universe (as presented in figure 8.1), but with the following differences:

1. We define a parameter *WordPress-require*:
   - the require arity of the port `wordpress` on the component type HTTP load balancer is equal the value of the parameter *WordPress-require*,
   - and the require arity of the port `wordpress` on the component type DNS load balancer is equal $2 \times$ *WordPress-require* + 1.

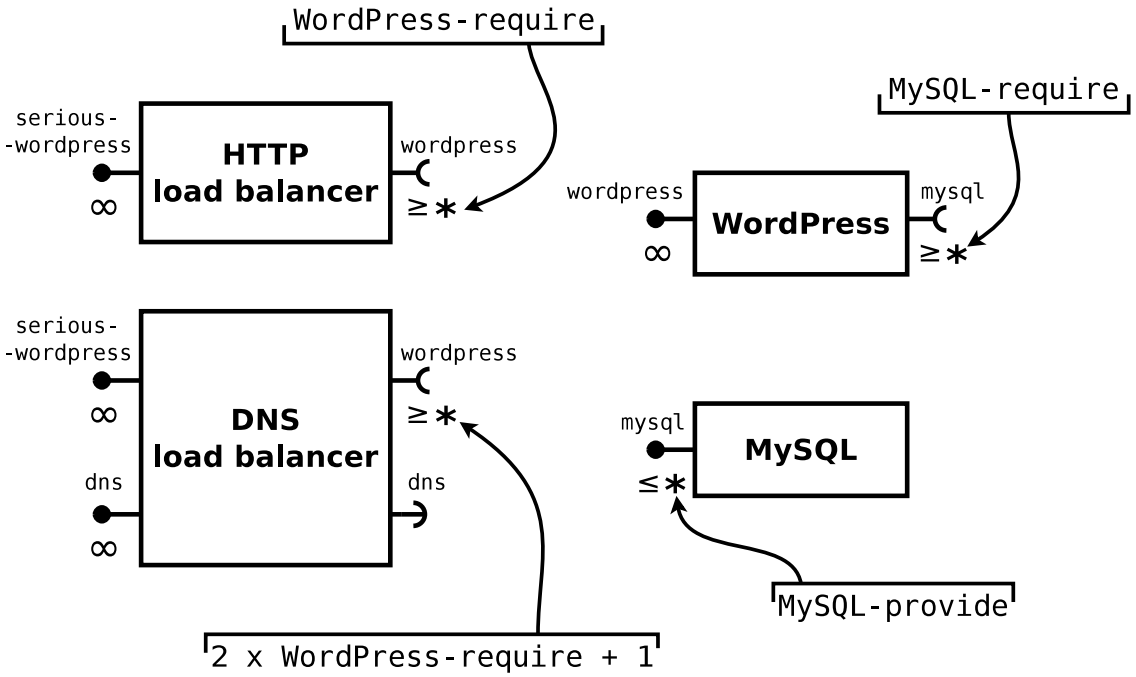


Figure 8.2: The parametrized stateful universe for the distributed WordPress use case.

2. We define a parameter *MySQL-require*:

   - the require arity of the port `mysql` on the component type WordPress is equal the value of the parameter *MySQL-require*.

3. Finally we define a parameter *MySQL-provide*:

   - the provide arity of the port `mysql` on the component type MySQL is equal the value of the parameter *MySQL-provide*.

The full parametrized universe is presented in figure 8.2. For these values of parameters:

$$\begin{cases} \textit{WordPress-require} & = & 3 \\ \textit{MySQL-require} & = & 3 \\ \textit{MySQL-provide} & = & 3 \end{cases}$$

we obtain our standard use case version of the universe (i.e. the one from figure 8.1).

Initial configuration, specification and optimization function In order to have all the necessary Zephyrus inputs, we must still define the initial configuration, the specification and the optimization function.

Our **initial configuration** contains just one location, called `Stub Location`, which has the `Stub Repository` installed on it, does not provide any resources, and has a cost of 1. This location corresponds in fact to the whole distributed system from the flat model point of view. As we did not introduce any element which might interfere with placing different components together on one machine, all the components present in the final configuration are simply always placed on this single location. This way we are simulating the flat architecture synthesis problem using the extended architecture synthesis problem.

The **specification** is very simple, as the only thing we want is to have a working WordPress farm deployed, which corresponds in our encoding at least one `serious-wordpress` port available in the final configuration:

$$\texttt{\#serious-wordpress} \geq 1$$

We could of course express this property of the system in a different way, e.g. ask for at least one instance of the two load balancer component types like that:

$$(\#\texttt{HTTP load balancer} \geq 1) \vee (\#\texttt{DNS load balancer} \geq 1)$$

As it does not really change anything, we stick to the simpler form based on ports.

Finally the choice of the the **optimization function** must be made. We could simply use our favourite *compact* optimization here, which would (lexicographically):

1. first minimize the cost of used locations,
2. then minimize the number of deployed components,
3. finally minimize the number of installed packages.

However, as the cost of the used locations and number of installed packages is always constant in our solutions (i.e. both are equal to 1 in every correct solution where the *WordPress-require* parameter is greater than 0), we have chosen to simplify the whole function and use only the middle optimization: minimizing the number of components. We call this optimization function *simple* and in fact it corresponds directly to the *compact* optimization function as it was defined for the flat model (see chapter 5, subsection 5.2.6).

All inputs together Summarizing, we call a *wordpress-flat* scenario corresponding to three parameters:

- *WordPress-require* $\in \mathbb{N}$
- *MySQL-require* $\in \mathbb{N}$
- *MySQL-provide* $\in \mathbb{N}$

an architecture synthesis problem instance with following inputs:

- a universe that contains exactly component types corresponding to the ones depicted in figure 8.2 with:
  - appropriate port require and provide arities replaced according to the provided formulas, using the values supplied for the three parameters,
  - a single repository `Stub Repository`, with a single package called the `Stub Package`,
  - an implementation relation is where each of the available component types can be implemented with one package: the `Stub Package`.
- an initial configuration with a single location: `Stub Location`,
- the following specification: `#serious-wordpress` ≥ 1,
- and an optimization function called *simple*, which is minimizing the number of components deployed in the final configuration.

The whole benchmark (i.e. the universe, initial configuration, specification and optimization function) corresponding to a problem instance associated with the parametrized scenario is automatically generated by Zephyrus (instead of reading all these inputs from external files) when given the following command line parameters:

```
./zephyrus -benchmark         wordpress
           -benchmark-option  wordpress_require  3
           -benchmark-option  mysql_require      3
           -benchmark-option  mysql_provide      3
```

These options designate the kind of benchmark which should be created and the benchmark generation options. If we do not specify all the options explicitly, the corresponding *wordpress-flat* scenario parameters are set to their default values (which is equal 3 for the three parameters introduced here).

8.2.2 Basic flat WordPress benchmarks

Let us explain our two first series of benchmarks based on the *wordpress-flat* scenario and comment on the results.

Variability We have tested the Zephyrus performances on the *wordpress-flat* scenario, while changing two of the three parameters:

- the *WordPress-require* parameter varies from 0 to 300 with a step of 20,
- and the *MySQL-require* parameter also varies from 0 to 300 with a step of 20.

The third parameter, *MySQL-provide*, is constant and always equals 3.

Moreover, we have conducted the same series of measurements using Zephyrus with two different constraint solvers:

- the standard constraint solver from the G12 project [112]
- and the Gecode constraint solver [105].

This gives two sets of 256 data points.

Time measurement For every combination of these parameters we have measured the time that Zephyrus takes to perform the whole execution run, from treating the inputs to generating and outputting the final configuration.

In fact the only difference from a standard Zephyrus run is what "treating the inputs" means for us here. We are not actually reading and parsing a series of input files found at the given paths in the file system, but instead we rather generate the universe, initial configuration and specification on-the-fly, using the Zephyrus built-in benchmark input generator module. Fortunately this changes the performances negligibly and does not affect the overall benchmark results.

The time measurements themselves are carried out by launching Zephyrus through the Linux `time` command (i.e. the `/usr/bin/time` [2]). More exactly what we take into account is the so-called *user time*: the total number of CPU-seconds that the Zephyrus process spent in user mode until exiting.

Presentation of the results The results of our benchmarks have been put in a graphical form using the python plotting library *matplotlib* [63]. We can see the effects in figure 8.3a (for the G12 solver) and figure 8.3b (for the Gecode solver). We have put:

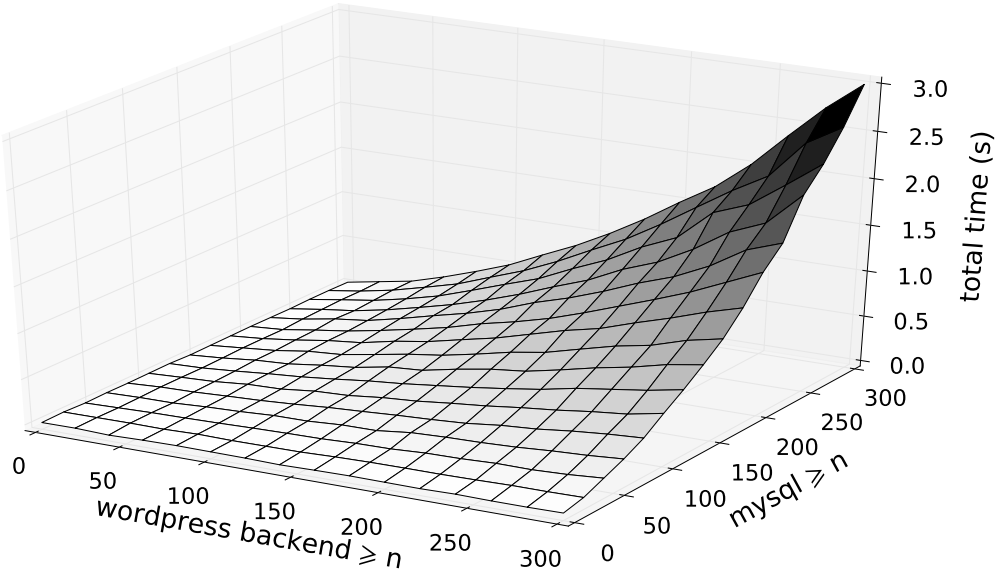
- the parameter *WordPress-require* on one horizontal axis (labelled "wordpress backend $\leq$ n"),
- the parameter *MySQL-require* on the other horizontal axis (labelled "mysql $\leq$ n"),
- and the elapsed user time in seconds on the vertical axis (labelled "total time (s)").

In the following we use a notation (X, Y) to refer to the data point on corresponding to the *WordPress-require* parameter equal X and the *MySQL-require* parameter equal Y on each chart.

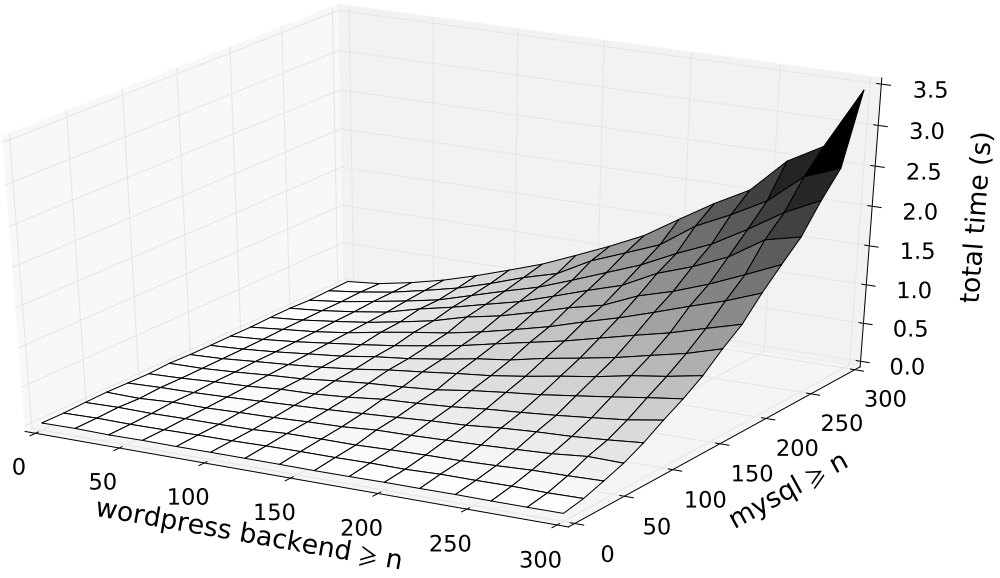
Interpretation of the results Both graphs are very smooth and look as expected. They show a steady rise in the Zephyrus execution times when the number of components and bindings in the final configuration is growing because of the higher benchmark parameters.

The minimal execution time, found at the data point $(0, 0)$, is very close to zero in both cases. The maximal time on the other hand, noted without surprise in both charts at the data point $(300, 300)$, varies following the used solver: G12 seems to fare slightly better, getting the maximal time around 3 seconds, while Gecode is a little bit slower and arrives there at 3 seconds and a half.

The conclusion which we can derive from these results is encouraging. The performance of Zephyrus for all the tested variants of the *wordpress-flat* benchmark is good and the execution times rise smoothly with the problem's size.



(a) Solver used: G12



(b) Solver used: Gecode

Figure 8.3: Simple `wordpress-flat` benchmarks.

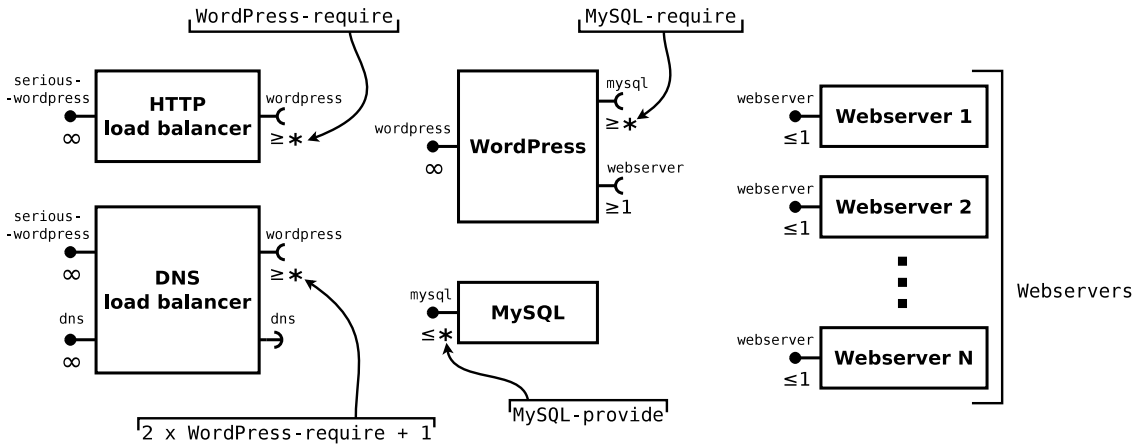


Figure 8.4: The parametrized stateful universe for the distributed WordPress use case extended with Webservers.

8.2.3 Flat distributed WordPress scenario with Webservers

Now, in order to perform some more interesting benchmarks, we extend our flat universe just a little bit.

Extending the universe We add some additional component types called Webservers. The number of different available component types is parametric and each component type has a name generated using a simple pattern: "Webserver 1", "Webserver 2", etc. Other than that, all of them are exactly identical: they simply provide one port `webserver`, with provide arity equal one. Also, we slightly modify the existing WordPress component by adding a new require port `webserver` with require arity equal one.

These modifications of the universe depend on a new benchmark parameter: *Webservers*:

- If the parameter *Webservers* is equal 0, then our universe remains exactly the same as previously;
- However if the parameter *Webservers* is a positive number n, then two changes happen:
  - We add n new component types to the universe: "Webserver 1", "Webserver 2", ..., "Webserver n". Each of them provides a single port `webserver` with provide arity 1.
  - We add a require port `webserver` with require arity 1 to the WordPress component.

The corresponding modified universe (for the value of the parameter *Webservers* at least equal one) is presented in figure 8.4. From now on we call it the "full *wordpress-flat* scenario" and treat the previously used scenario version as its special case for the parameter *Webservers* equal to 0.

Aim The principal aim of adding the whole family of Webserver component types to our universe is to introduce an easily controllable and scalable amount of symmetries into our problem.

As each WordPress instance requires one binding on the `webserver` port and each Webserver instance can provide maximally one such binding, we obviously have exactly as many Webserver instances in our final configuration as WordPress instances. For each of these instances the choice among different available Webserver component types (i.e. an instance of which particular component type should be used) is completely arbitrary, as they are all effectively identical on the model level.

From the constraint solving point of view this corresponds to a constraint problem with many symmetric solutions which are all equally correct and optimal and there is no reason whatsoever to choose one over the other. We can control the amount of these symmetric solutions very directly by changing the *Webservers* benchmark parameter.

The number of correct and optimal solutions of a given scenario with the parameter *Webservers* equal 0 (i.e. like in all our previous benchmarks) grows quickly when we increase this parameter. If we look naively at the problem, it may seem that it even grows exponentially (choosing one Webserver among a fixed number of available ones for each WordPress instance would correspond to number of options equal to *Webservers* to the power of *WordPress-require*), but as on the constraint problem level the components are treated *en masse*, the speed of growth is much smaller than that: we distribute a certain number of identical WordPress components into a given number sets, which gives us exactly

$$\frac{(w+s-1)!}{w!\,(s-1)!}$$

possibilities when we take:

- w equal to the value of the *WordPress-require* parameter,
- s equal to the value of the *Webservers* parameter.

Meaning These Webserver components are mainly added in order to introduce symmetries in our *wordpress-flat* scenario, so that we can measure how the constraint solvers which we use are able to handle them.

We can imagine that this represents a choice among several different web servers (e.g. Apache2 [113], Ngix [91] or lighthttpd [71]) which we could used to implement a part of the service corresponding to each deployed Wordpress component.

Nevertheless, we should mention that, in practice we would never encode such a design decision in the Zephyrus model in this way. There are various reasons not to do that, but above all there is strictly no point in making Zephyrus decide between multiple alternatives which are completely indistinguishable at the model level.

8.2.4 Advanced flat WordPress benchmarks

Let us now present and explain a more comprehensive series of benchmarks, based on the full *wordpress-flat* scenario (i.e. the previous scenario extended with the family of the Webserver component types).

Variability We have tested the Zephyrus performances on the full *wordpress-flat* scenario, while changing several parameters:

- the *WordPress-require* parameter varies from 0 to 100 with a step of 20,
- the *MySQL-require* parameter varies from 0 to 100 with a step of 20, with an additional data point at *MySQL-require* = 1,
- the *MySQL-provide* parameter takes two values: 3 and 4,
- and finally the *Webservers* parameter varies from 0 to 5.

As before, we have conducted the same series of measurements using Zephyrus with two different constraint solvers:

- the standard constraint solver from the G12 project [112]
- and the Gecode constraint solver [105].

Moreover, this time we have also executed Zephyrus for in each case in two different modes:

- the standard mode,
- and the `stop-after-solving` mode, in which Zephyrus does not actually generate the final configuration, but exits directly after solving the constraint problem.

This us gives in total 2016 data points.

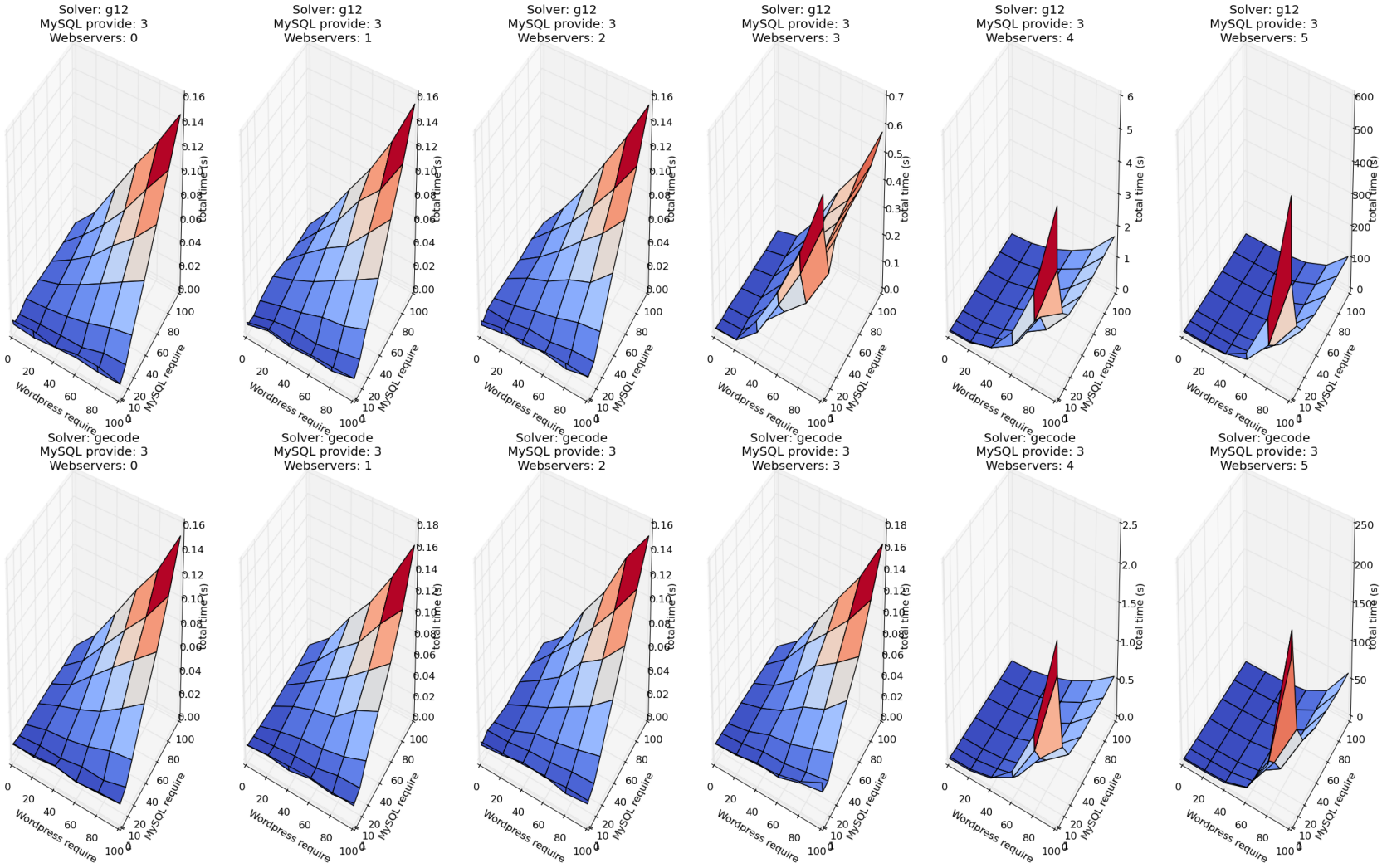


Figure 8.5: A series of `wordpress-flat` benchmarks with changing `webservers` parameter, solvers used: G12 / Gecode

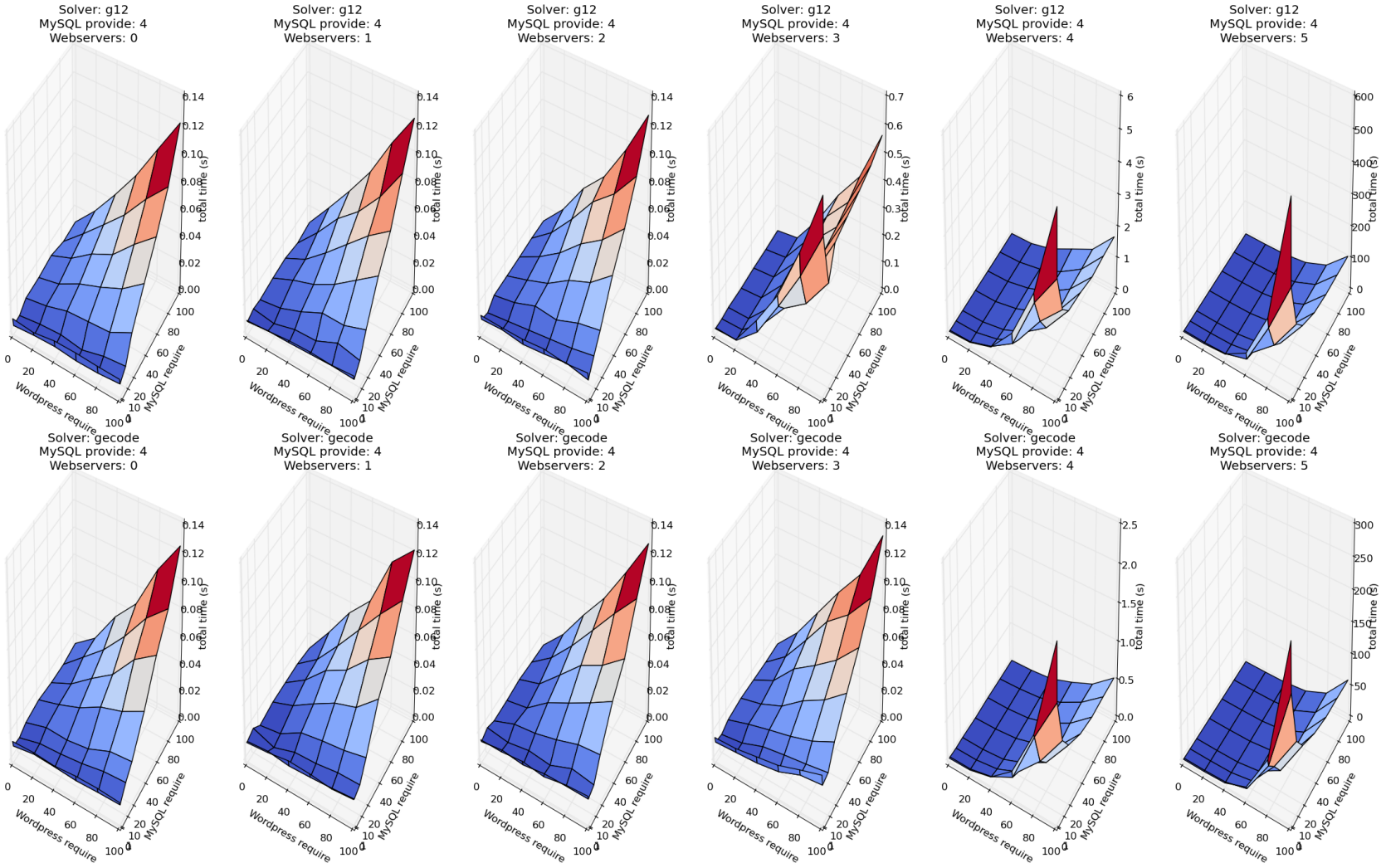


Figure 8.6: A series of `wordpress-flat` benchmarks with changing `webservers` parameter, solvers used: G12 / Gecode

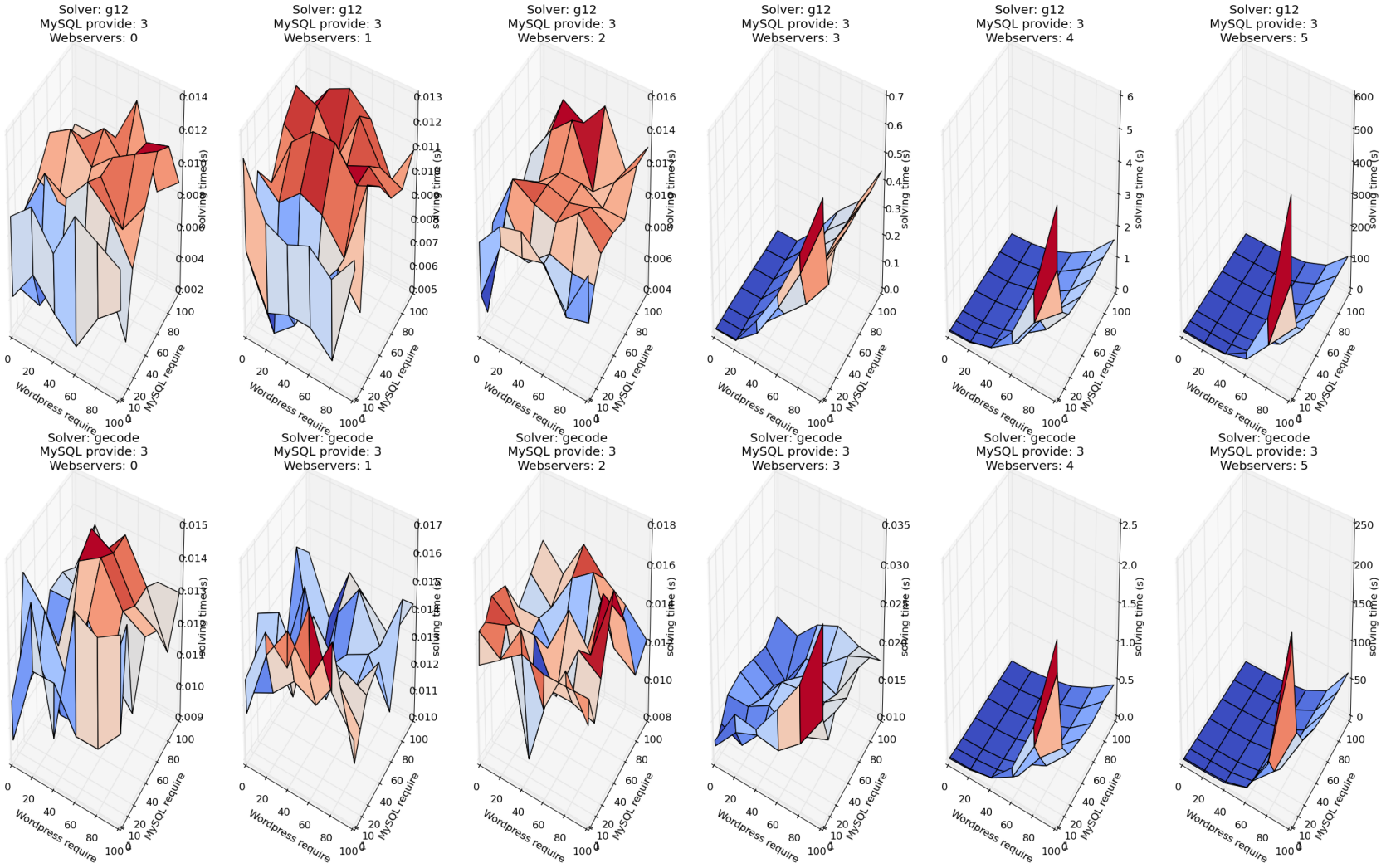


Figure 8.7: A series of `wordpress-flat` benchmarks with changing `webservers` parameter, solvers used: G12 / Gecode, no configuration generation

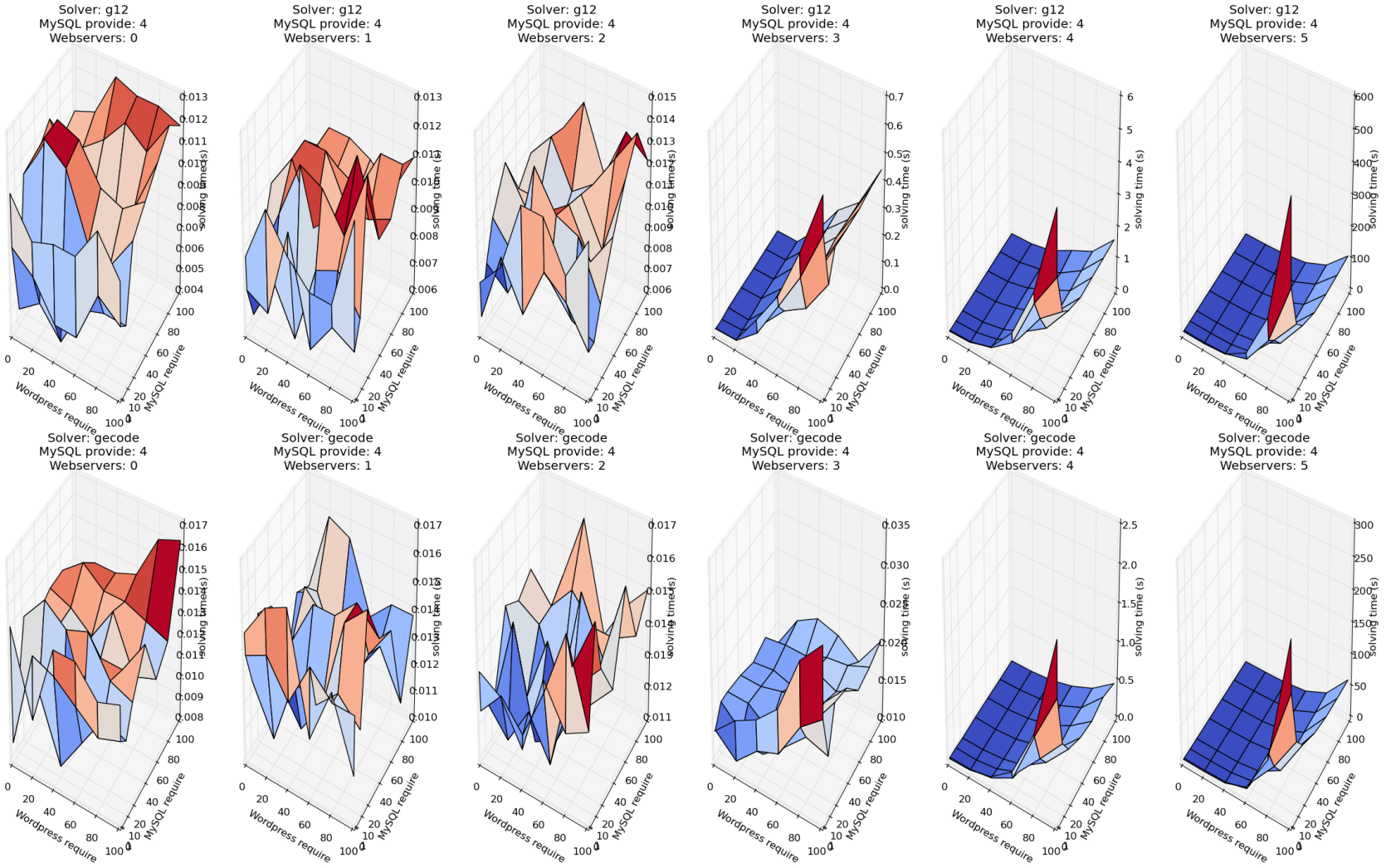


Figure 8.8: A series of `wordpress-flat` benchmarks with changing `webservers` parameter, solvers used: G12 / Gecode, no configuration generation

Presentation of the results As this time we have many more problem dimensions to consider, we need quite a large number of individual graphs to present all this information. Therefore we divide all the data points in 4 pages (series) of 12 graphs in the following way:

- On each individual graph we have:
  - the parameter *WordPress-require* on one horizontal axis (labelled "Wordpress require"),
  - the parameter *MySQL-require* on the other horizontal axis (labelled "MySQL require"),
  - and the elapsed user time in seconds on the vertical axis (labelled "total time (s)" or "solving time (s)").
- On each page (series) we have:
  - Two horizontal lines of graphs, one for each solver: Gecode (the bottom line) and G12 (the top line),
  - and in each line the `Webservers` parameter increases from left to right.
- And we have four series of benchmarks on four different figures (put on four separate pages for better readability):
  1. figure 8.5 corresponds to the parameter `MySQL-provide` equal 3 and each graph shows the total Zephyrus execution time,
  2. figure 8.6 corresponds to the parameter `MySQL-provide` equal 4 and each graph shows the total Zephyrus execution time,
  3. figure 8.7 corresponds to the parameter `MySQL-provide` equal 3 and each graph shows the Zephyrus execution time without generating the final configuration,
  4. figure 8.8 corresponds to the parameter `MySQL-provide` equal 4 and each graph shows the Zephyrus execution time without generating the final configuration.

As before, in the following we use a notation (X, Y) to refer to the data point on corresponding to the *WordPress-require* parameter equal X and the *MySQL-require* parameter equal Y on each chart.

Interpretation of the results There are many interesting facts which we can read in these benchmark results:

1. First quite trivial observation might be that, the graphs showing the total Zephyrus execution time for the low *Webservers* parameters (i.e. *Webservers* ≤ 2) are very similar to our previous results.

   Of course this was entirely expected, as the sets of scenarios tested in these cases are very close to the ones tested before. The major difference is the previous graphs went up to 300 for both parameters.

   Also, as we can see, the change from 0 to 2 Webserver component types does not seem to have any significant effect on the Zephyrus performances:

   - on the one hand, as there is exactly no difference in the number of our constraint problem's optimal solutions when we go from *Webservers* = 0 to *Webservers* = 1, the lack of change in performance here is not a big surprise;
   - on the other hand, when we go from *Webservers* = 1 to *Webservers* = 2 the number of optimal solutions is multiplied by the number of deployed WordPress components (plus 1), which is thus maximally equal 100 (i.e. the maximum of our *WordPress-require* parameter), that apparently is negligible for the constraint solver.

   The difference between the performances of our two solvers is also not noticeable.

2. When we compare the mentioned set of graphs (i.e. the ones for the low values of the *Webservers* parameter: *Webservers* ≤ 2) with the corresponding graphs showing the the Zephyrus execution time without generating the final configuration (i.e. the `stop-after-solving` mode) we can see something very interesting. When Zephyrus does not need to generate the final configuration, its execution times get much lower and in fact quite random, as they do not really seem to be correlated with the *WordPress-require* and *MySQL-require* parameters.

   The only reasonable conclusion which we can draw from this observation is that for these cases the time of solving the constraint problem is negligible in comparison with the time needed to generate the final configuration. It may seem surprising, but actually it is not so strange: our constraint problems corresponding to all these scenarios are, as a matter of fact, rather simple, while the number of bindings to generate is very big in comparison (for the case $(100, 100)$ without any Webserver components we have 100 WordPress instances, each requiring 100 bindings with MySQL instances, which gives a total of 10.000 bindings to generate).

   In spite of this, most of the observations that we have already made still hold, because on these graphs we do not see any significant difference between the results for two tested solvers and when changing the *Webservers* parameter between 0 and 2.

3. Another trivial observation is that there is no discernible difference between the same graphs corresponding to the two values of the *MySQL-provide* parameter: 3 and 4. In fact the change in this parameter was introduced to our benchmarks only as a kind of a sanity check, to see if there are no unexpected things happening when we play with its value just a little.

4. Now let us cover all the more interesting and surprising observations concerning these benchmark results. We start with mentioning that systematically on many graphs we can observe a strange fact that for the parameter `MySQL-require` equal 0 we get higher execution times than for for `MySQL-require` equal 1. We would expect it to be the other way around, as it seems that the constraint problems corresponding to the value 0 of this parameter (where WordPress components do not require any bindings at all and thus the MySQL components are strictly useless) should be easier than those corresponding to 1 or higher.

5. As we would predict, the execution times rise greatly with higher values of the *Webservers* parameter (i.e. *Webservers* ≥ 3). In fact we can see a sharp change in the graph's form occurring between:

   - *Webservers* passing from 2 to 3 for the G12 solver,
   - *Webservers* passing from 3 to 4 for the Gecode solver.

   That particular difference between the two solvers is probably related to how well each of them handles the kind of solution symmetries introduced by our Webserver components. This also seems to be part of a more general trend, as the Gecode solver performs in most cases better than the G12 solver.

   However, there is another very surprising thing happening in the graphs for the high *Webservers* parameter values: the position of the peak (i.e. the case where the execution time is the highest). Instead of finding it, as usual and expected, in the right corner of the graph (at the highest `WordPress-require` and the highest `MySQL-require`), it is rather systematically present in its bottom corner (at the highest `WordPress-require` and the lowest *Webservers*).

   There are several theories which would explain this misplaced peak, but we have no hard evidence for any of them. Either way it seems that, counter-intuitively, the constraint problems corresponding to the smallest values of the *Webservers* parameter are more difficult than those corresponding to the other values. The main conclusion that we can draw from the existence of this *artefact* is that apparently the constraint solvers can indeed be unpredictable and changing the shape of a supposedly simple problem in some way does not necessarily have to be linked in a clear fashion with the change of the problem's solving time.

8.2.5 Fully distributed WordPress scenario

Now we pass to the fully distributed WordPress scenario and present all the benchmarks based on it. The fully distributed WordPress scenario is similar to our flat WordPress scenario without the Webserver component types, but completed with all the Zephyrus model features related to component placement. Our component types, initial configuration and specification are modified in order to add the relevant information.

Repositories and packages The part of our scenario related with the package repositories (as well as the component type implementation relation) remains unchanged.

We could add a big package repository, based on a real-world one, to our universe and we could make every component type implemented by a separate package, but in fact it does not make a lot of sense. We want to have only one repository in this scenario (i.e. one Linux distribution) and the four services represented by our component types are all co-installable with each other without problem (at least in most non-obscure Linux distribution). Thus there is simply no point in bringing packages the into the game, for two complementary reasons:

- Packages are meant to model the installability of components on locations and the local conflicts between components and both these issues are absent from the considered scenario. So we can simply omit including the packages in it, because they add nothing of value: they do not influence the component part of the solution in any way.
- We could actually include a big package repository (e.g. over 40.000 packages of a Debian Linux distribution) in our universe and modify the implementation relation accordingly. But then applying our advanced package trimming would leave us in the best case with four individual packages (all not connected in any way and always installable everywhere). Which is equivalent to having simply one stub package.

Resources We include one resource in the fully distributed WordPress scenario, it is called `ram` and it represents roughly the amount of RAM memory used by the running services corresponding to our components.

The default resource consumption of each component type was chosen arbitrarily, mostly in order to obtain right proportions between our components, but it is still quite reasonable. Our scenario is parametric though, so these values can be adapted at will:

Component Type	Parametric value	Default
DNS load balancer	*DNS-consume*	64
HTTP load balancer	$8 \times$ *DNS-consume*	512
WordPress	*WordPress-consume*	512
MySQL	*MySQL-consume*	512

As we can see there are three parameters which set the resource consumption of our component types directly: *WordPress-consume*, *MySQL-consume* and *DNS-consume*. The resource consumption of our fourth component type, HTTP Load Balancer, is fixed at 8 times the consumption of the DNS Load Balancer

Let us recall, that the choice of the load balancing component is supposed to be kind of a trade-off here: the DNS-based load balancer is much simpler, smaller and less resource-greedy, but it needs to have more WordPress backends (exactly 2 times more plus one) behind him in order to provide the same quality of service as the HTTP-based one, which on the other hand requires more computing power (exactly 8 times more), as it is performing caching.

Specification The part of our specification which expresses the user request stays the same as before. However, we add a new part which encodes a custom deployment policy: on each location there can be at most one WordPress instance and at most one MySQL instance. For example, we can have locations with where one WordPress instance and one MySQL instance are co-located, but a situation where two WordPress instances put together on a single location is not permitted.

In order to express this deployment policy in our specification syntax we have to use a slightly more complicated construction. In fact we encode it as two separate properties:

1. "the number of locations where there is more than one WordPress instance is equal zero",
2. "the number of locations where there is more than one MySQL instance is equal zero".

Thus our whole specification looks like that:

$$\texttt{\#serious-wordpress} \geq 1 \wedge$$
$$\#(\_)\{\texttt{Stub Repository} : \#\text{WordPress} > 1\} = 0 \wedge$$
$$\#(\_)\{\texttt{Stub Repository} : \#\text{MySQL} > 1\} = 0$$

Machine park Finally, in order to make our fully distributed WordPress scenario work, we must provide enough locations for our components to be placed on. Thus we must remove the stub location from our initial configuration, and put some actual locations instead.

We want to make our machine park model a public cloud and be more or less realistic, so we base the types of the available machines it on a little part of the "old" Amazon EC2 offer [108]:

Machine type name	`ram` provided	Cost
Old Small	1825	65
Old Medium	4026	130
Old Large	8052	260
Old Xlarge	16104	520

We adjust the size the set of locations actually included in the initial configuration using benchmark parameter *Park-size* (default is 40). The composition of this set is determined automatically, by putting an equal number of each machine type inside (thus the provided *Park-size* parameter values should be divisible by 4). Locations are named simply "Old Small 1", "Old Small 2", etc.

8.2.6 First series of distributed WordPress benchmarks

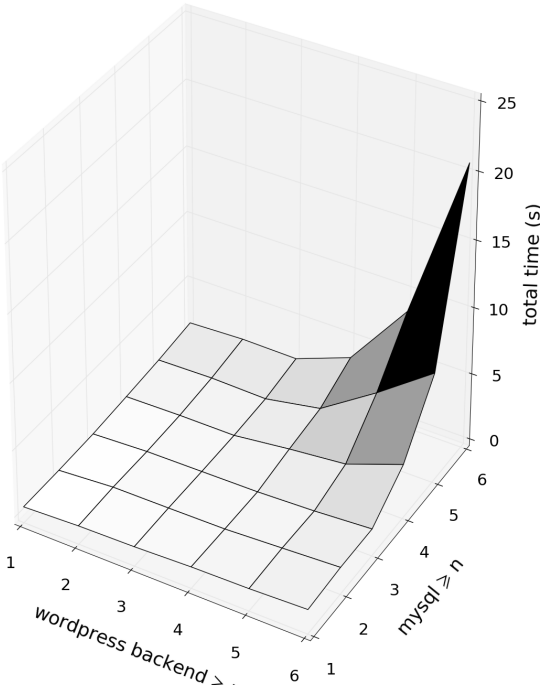
Let us now present the first series of benchmarks based on the *wordpress-dist* scenario with a restricted machine park.

Variability We have tested the Zephyrus performances on the *wordpress-dist* scenario, while changing two parameters:

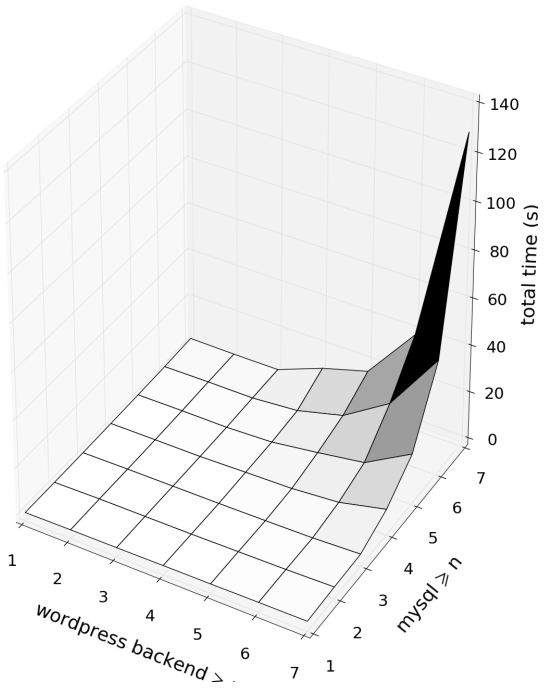
- the *WordPress-require* parameter varies from 1 to 10 with a step of 1,
- and the *MySQL-require* parameter equally varies from 1 to 10 with a step of 1.

Everything else than these two parameters are fixed. We are using only the Gecode solver here, which was more or less an arbitrary choice between Gecode and G12[1]. The machine park is fixed at the default size of 40 locations: 10 of each of the defined types.

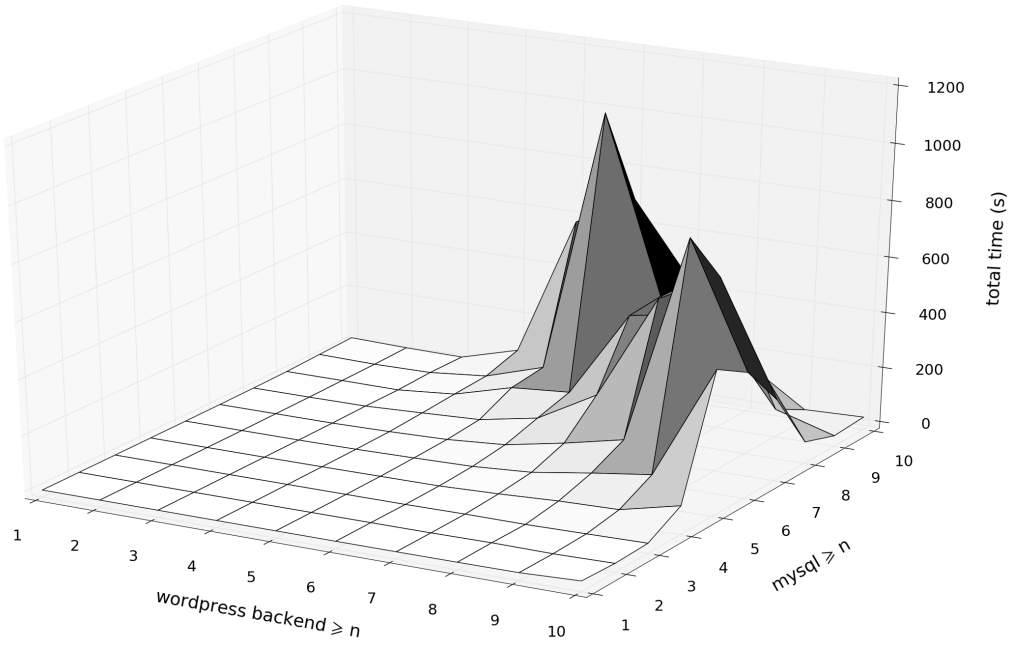
[1] In most cases results given by Gecode and G12 solvers are comparable and we simply do not need two series of nearly identical results here.



(a) Up to (6,6)



(b) Up to (7,7)



(c) Up to (10,10)

Figure 8.9: Three views of the same benchmark: two restricted, one full.

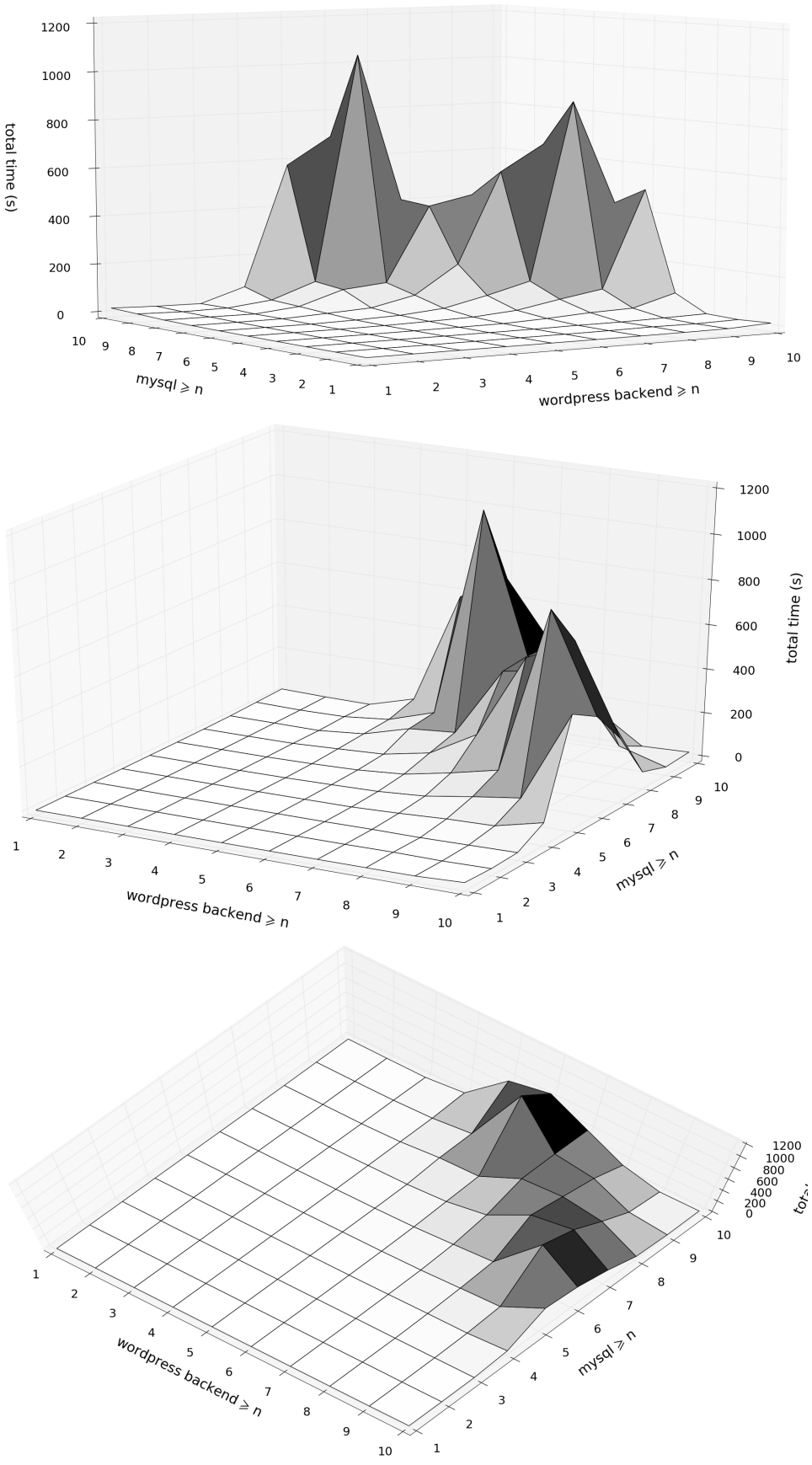


Figure 8.10: Three perspectives on the same benchmark diagram.

Presentation and interpretation of the results This series of benchmarks is much easier to present than the previous one, as there are only 100 data points. Nevertheless, there are several interesting perspectives that we can take to look at the graph we present here.

The whole graph is fully visible from three different angles in figure 8.10. We can clearly divide it in three separate areas of interest:

- first there is a large area of seemingly easy problem instances corresponding to the low values of both parameters,
- then we see something that resembles a chain of peaks going diagonally through the graph, which apparently the difficult problem instances, corresponding to a certain consistent set of combinations of the two parameters,
- and finally we see the problem difficulty going back down after traversing this "mountainous" area.

In order to gain some perspective about the seemingly flat area of easy problem instances, we should look on figure 8.9a and figure 8.9b. These two correspond exactly to the same data, simply the graph is cut after the $(6, 6)$ point or $(7, 7)$ point and the scale is adjusted accordingly. What we can notice is that what appears to be a flat area on the full graph is in reality quite steep! Simply, as the values for the difficult cases are an order of magnitude bigger, they make this part of the graph look almost flat in comparison.

Now let us explain the reason behind the fact that our graph looks like a model of a mountain chain. Intuitively, with higher parameter values we should systematically get more difficult problems which require more time to solve. In this benchmark series this is not the case, as our problem size is in fact restricted directly by the relatively small size of our machine park. Actually already for the case $(11, 11)$ our problem has no solution: it is not possible to fit all the necessary components on the available 40 locations.

Therefore finally the shape of our graph is not very surprising:

1. With small values of parameters the problem to solve is quite simple, as we have not so much components to place: the search space is easy to explore.
2. Then we arrive at the difficult instances, where we have many components to place and a lot of locations to put them on in various different combinations: the search space that needs exploring becomes bigger and bigger.
3. After a certain point the number of components becomes so big, that our machine park becomes crowded and there are not so many different possibilities of placing components on the machines any more: the search space becomes reduced.
4. Finally, at the point $(11, 11)$ there are simply too much components and no way to place them on: the solution space is empty.

8.2.7 Machine park trimming benchmarks

The next step on our road is a series of benchmarks which check if we can efficiently simulate a cloud in our approach by using huge machine machine parks and reducing their size automatically (for each given scenario) with our location trimming methods. Proving this lets us continue the benchmarks on the fully distributed WordPress scenario with a less restricted machine park.

Variability Here we stick almost to the same version of the *wordpress-dist* scenario all the time:

- the *WordPress-require* parameter is equal 4,
- and the *MySQL-require* parameter is equal 4,
- all the other parameters have default values.

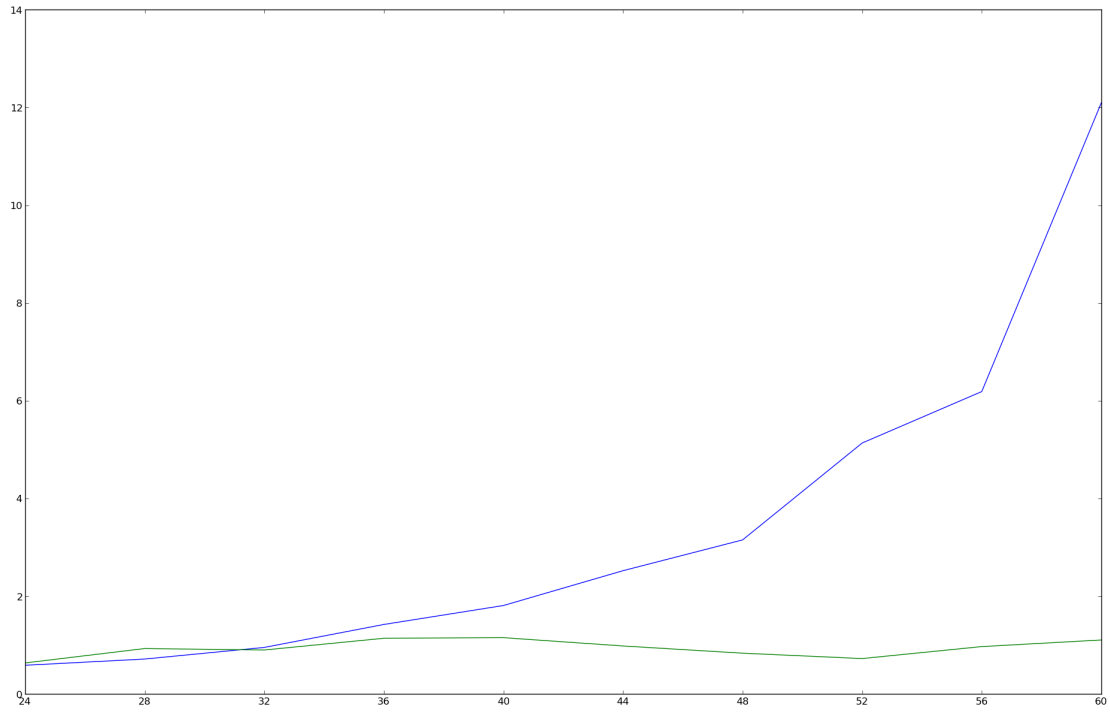


Figure 8.11: Effect of location trimming: up to 60 locations.

- Also, we stick to one solver (chosen arbitrarily): Gecode.

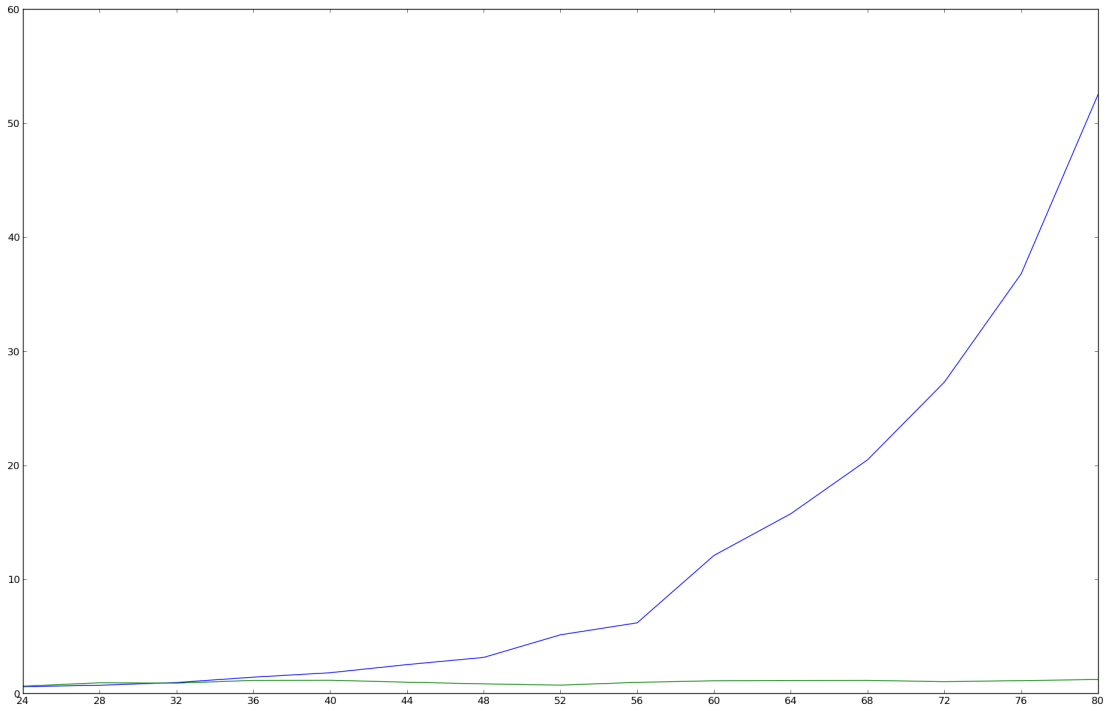
The only actual benchmark parameter that varies is *Park-size*, which starts at 24 (it is the minimal number of machines necessary to deploy our scenario optimally, i.e. only on Small machines) and goes up to 100 with a step of 4. Our purpose is to investigate what happens when we increase this parameter, thus getting bigger and bigger initial machine parks, and turn off or on the Zephyrus location trimming feature (which corresponds in practice to running Zephyrus with the option `-use-all-locations off` or `-use-all-locations on`).

Presentation of the results In fact we have only one graph to show here, but we have prepared three versions of it in order to see more clearly what is happening. All three are constructed in the same way:

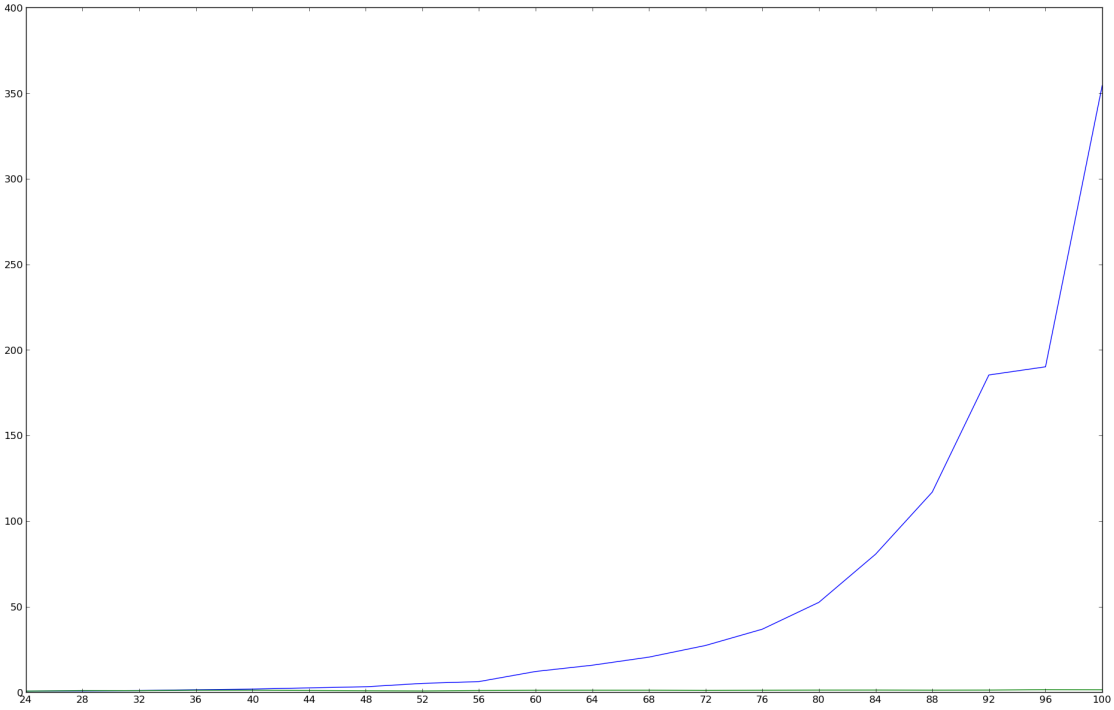
- on the horizontal axis we have the parameter *Park-size*, starting at 24 and going up;
- on the vertical axis we have the total Zephyrus execution time in seconds;
- and the correlation between the machine park size and the execution time is represented by the two lines:
  - the blue line shows what happens when the location trimming is **off**,
  - and the green line shows what happens when the location trimming is **on**.

The only thing which changes is where the horizontal axis ends, the maximal *Park-size* value is equal to:

1. 60 in figure 8.11,
2. 80 in figure 8.12a,
3. 100 in figure 8.12b.



(a) Up to 80 locations.



(b) Up to 100 locations.

Figure 8.12: Larger cases confirm that location trimming is advantageous.

Interpretation of the results In all three of these cases we can clearly see, that the the machine park trimming becomes more and more advantageous when the difference between the actual size of the machine park and the minimal size needed to find an optimal solution (i.e. 24 locations) grows. When these two are close (i.e. there are not too many superfluous locations present), as we can notice when examining the values in figure 8.11 which correspond to *Park-size* between to 24 and 36, turning on the location trimming seems to make things slightly worse or not to help a lot. However, with growing *Park-size*, the gap between the two lines becomes bigger and bigger.

There are two causes why having superfluous locations in our configuration is so disadvantageous and hence why the location trimming seems to be so profitable:

- First, adding a new location to a machine park corresponds effectively to multiplying the amount of the local variables (i.e. $N(l, x)$) related with component types, ports and packages, the repository variables (i.e. $R(l, r)$) and all the associated constraints. And this intuitively makes our constraint problem heavier and heavier.
- Second, as the number of locations grows, the number of symmetric solutions to a given scenario grows with it. Even though we try to introduce some constraints which remove the component placement symmetries, it is very difficult to take care of all of them.

On the other hand it seems, especially when we look on figure 8.12b, that our location trimming method scales quite well. Each point marked by the green line corresponds to solving the same main constraint problem, as the initial configuration is trimmed in every case to the same set of 24 locations. Therefore the difference between the points can come solely from the time spent on performing the pretreatment, namely trimming the locations.

The conclusion that we can draw from this series of benchmarks is, that we should be able to easily simulate a cloud simply by using a big number of available locations together with our location trimming mechanism (at least when our machine park is structured similarly to the the one considered here).

8.2.8 Advanced distributed WordPress benchmarks

Now we repeat the fully distributed WordPress scenario benchmarks presented before, but with two important improvements:

- we are using some more powerful solving techniques (namely the G12/CPX solver and the portfolio approach)
- and we include a much bigger machine park in order to simulate a cloud environment better.

Solving techniques In this series of benchmark we change the methods that we use to solve constraint problems:

- first we use the G12/CPX solver instead of the standard G12 solver or Gecode which were employed in all the previous tests,
- then we combine these three solvers (i.e. G12 standard, Gecode and G12/CPX) into a single constraint solving mechanism based on the portfolio method (i.e. basically launching several solvers simultaneously on the same problem instance).

Thanks to these techniques, we are capable of going in this series of benchmarks up to the point $(16, 16)$ without exceeding the execution time of 30 minutes.

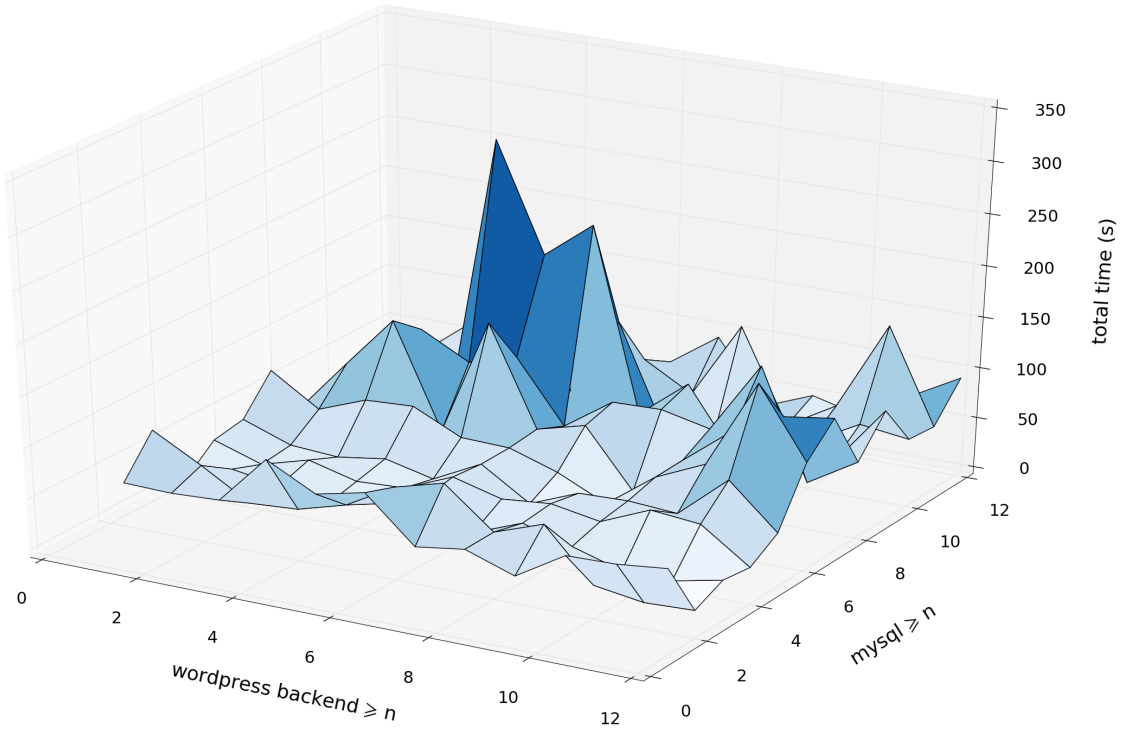


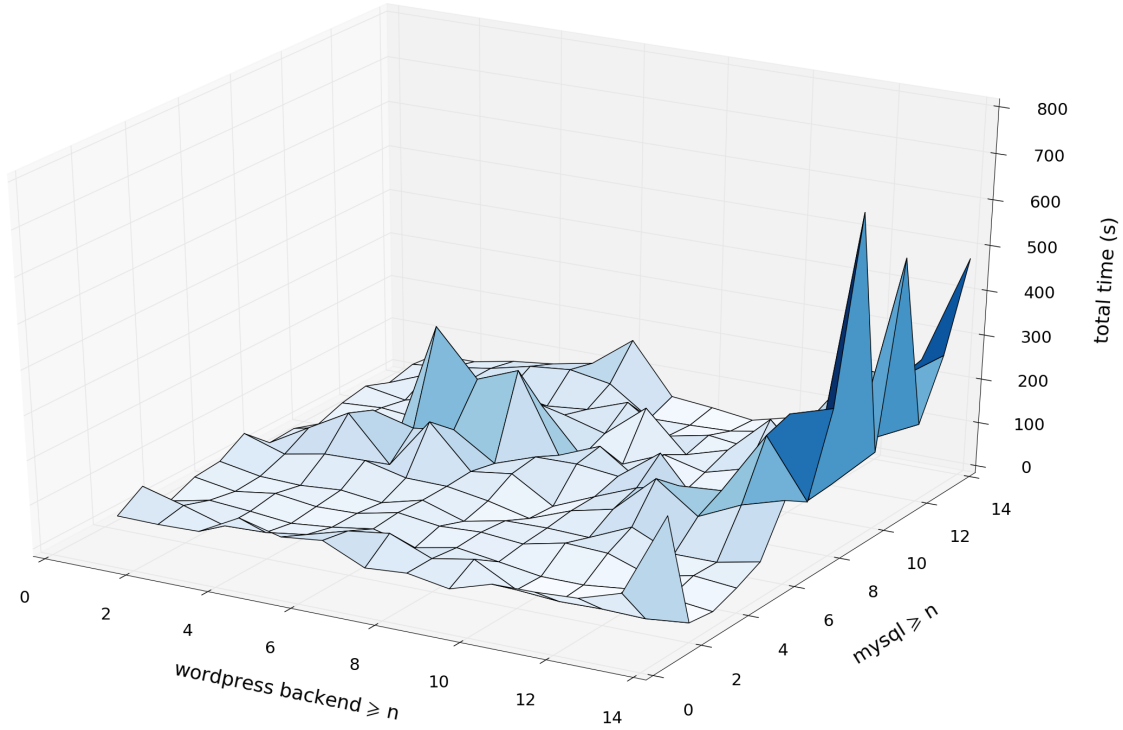
Figure 8.13: CPX up to (12,12).

The machine park Instead of a relatively small machine park of 40 machines, we now always provide Zephyrus with a considerable machine park of 1000 machines. In other words, the value of the *Park-size* parameter is fixed at 1000 in all the following examples.

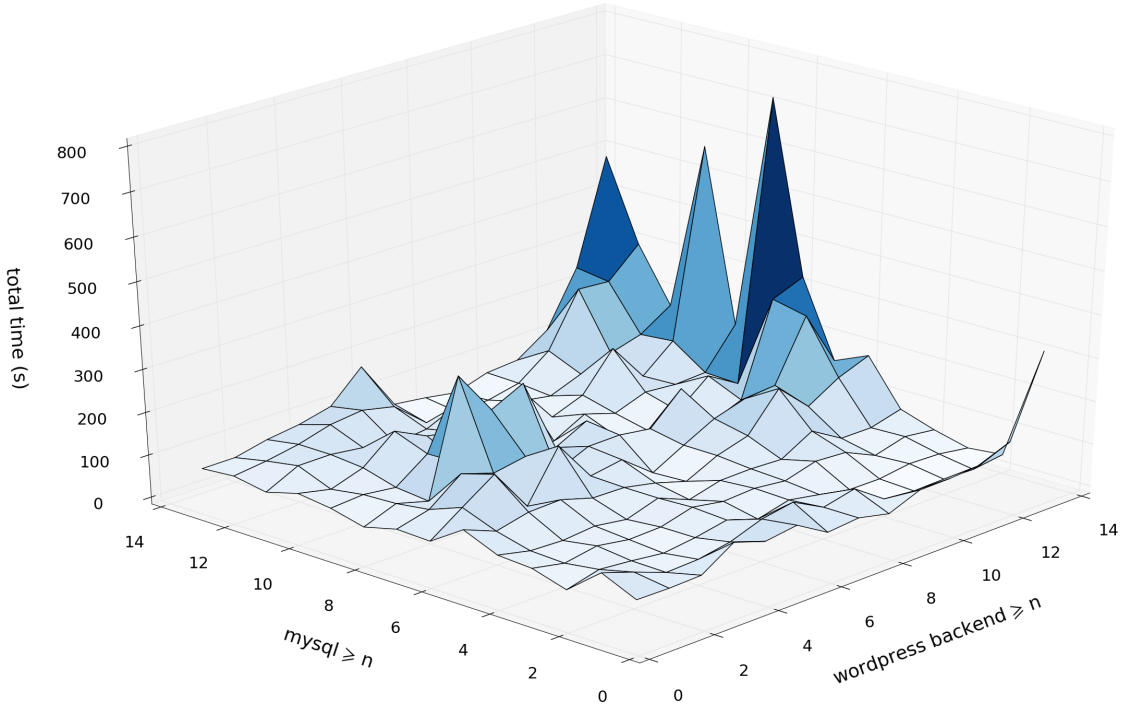
As we know, thanks to our location trimming methods the actual set of locations included in the solved constrain problem is much smaller than that. Yet it is always such a subset of our 1000 machines that allows Zephyrus to find a solution as optimal as the solution for the whole set of 1000 machines would be.

Presentation of the results Similarily to the previous distributed WordPress scenario benchmarks, here we have only two graphs, which are presented from various different perspectives:

- The first graph corresponds to using the G12/CPX solver in all the scenario variants. There are three sub-graphs that show it cut at various points:
  - in figure 8.13 it goes up to the point (12, 12),
  - in figure 8.14 it goes up to the point (14, 14),
  - in figure 8.15 it goes up to the point (16, 16), which is the maximum.
- The second graph corresponds to using the portfolio solving method and there are again three sub-graphs that show it cut at various points:
  - in figure 8.16 it goes up to the point (12, 12),
  - in figure 8.17 it goes up to the point (14, 14),
  - in figure 8.18 it goes up to the point (16, 16), which is the maximum.

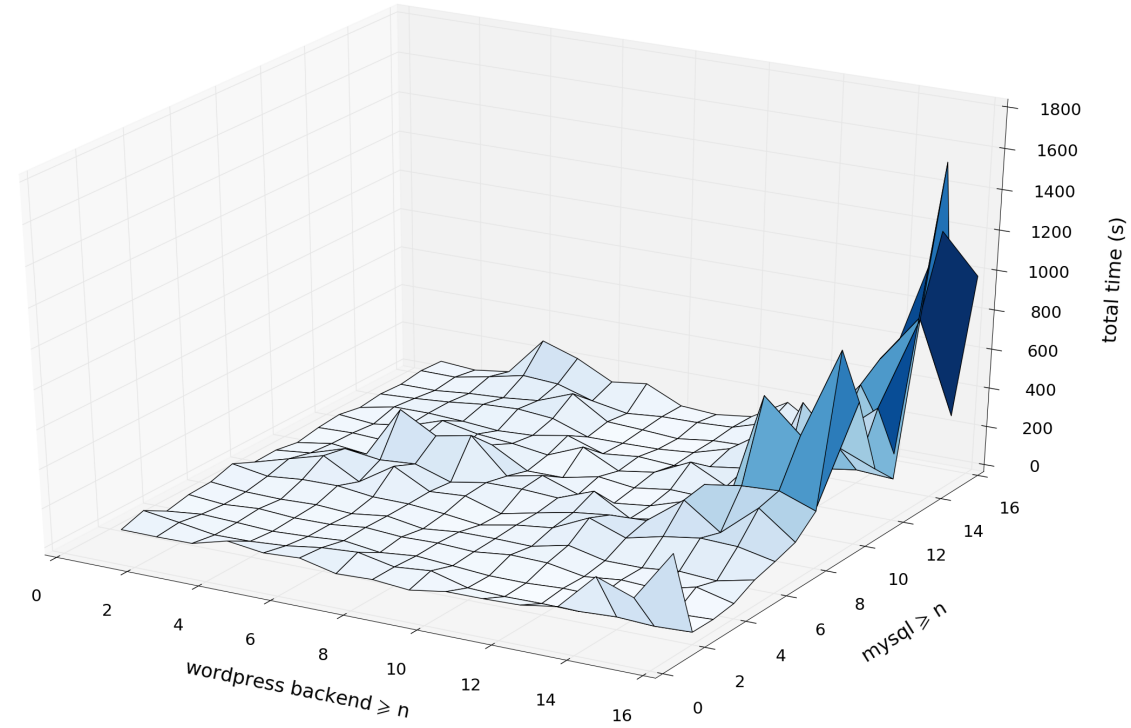


(a) Standard angle.

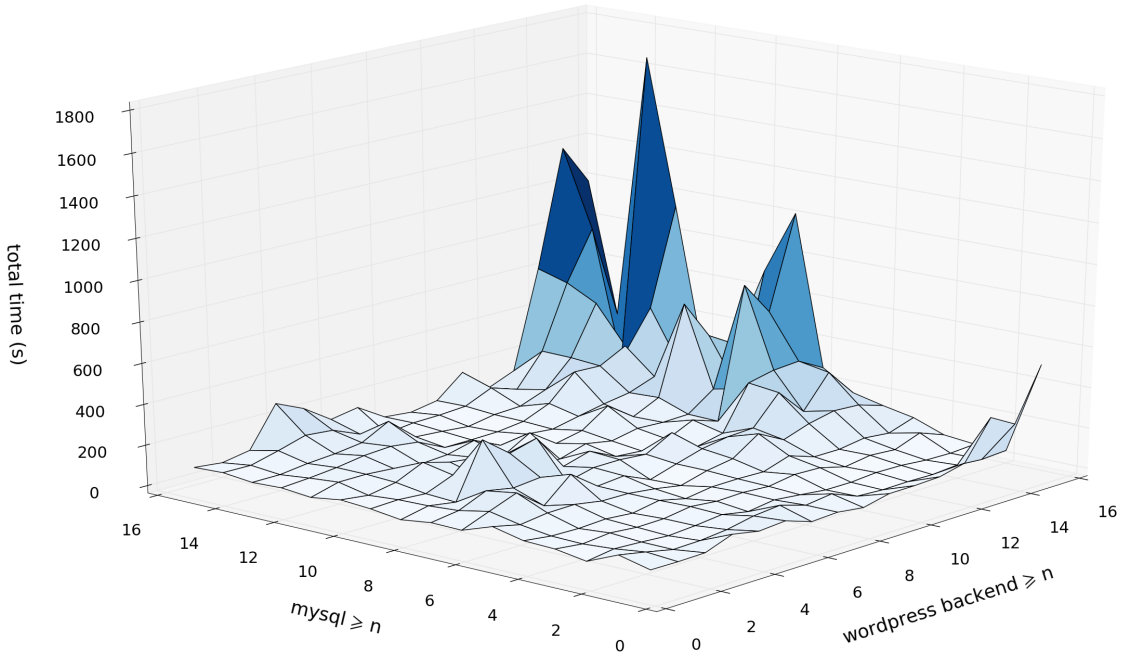


(b) Different angle.

Figure 8.14: CPX up to (14,14).



(a) Standard angle.



(b) Different angle.

Figure 8.15: CPX up to (16,16).

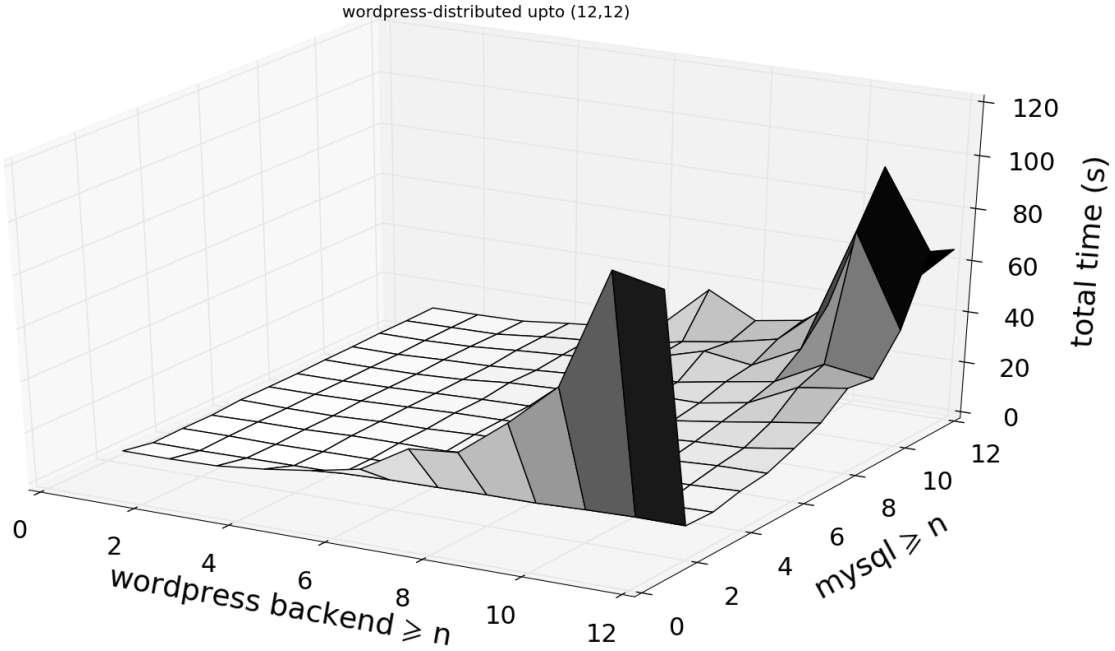


Figure 8.16: Portfolio up to (12,12).

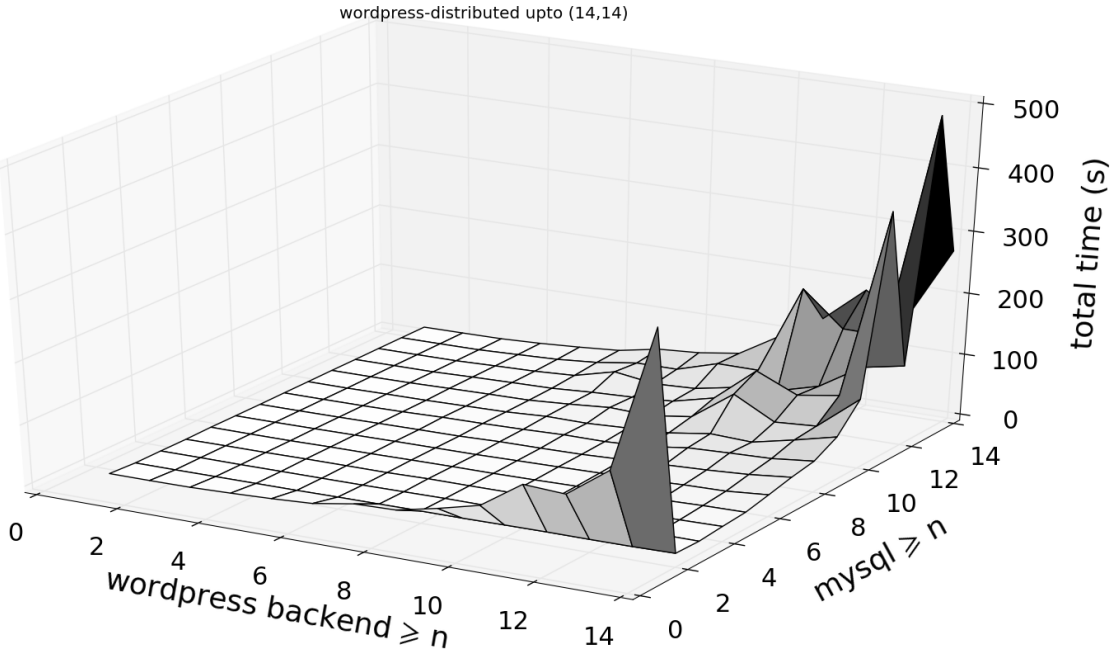


Figure 8.17: Portfolio up to (14,14).

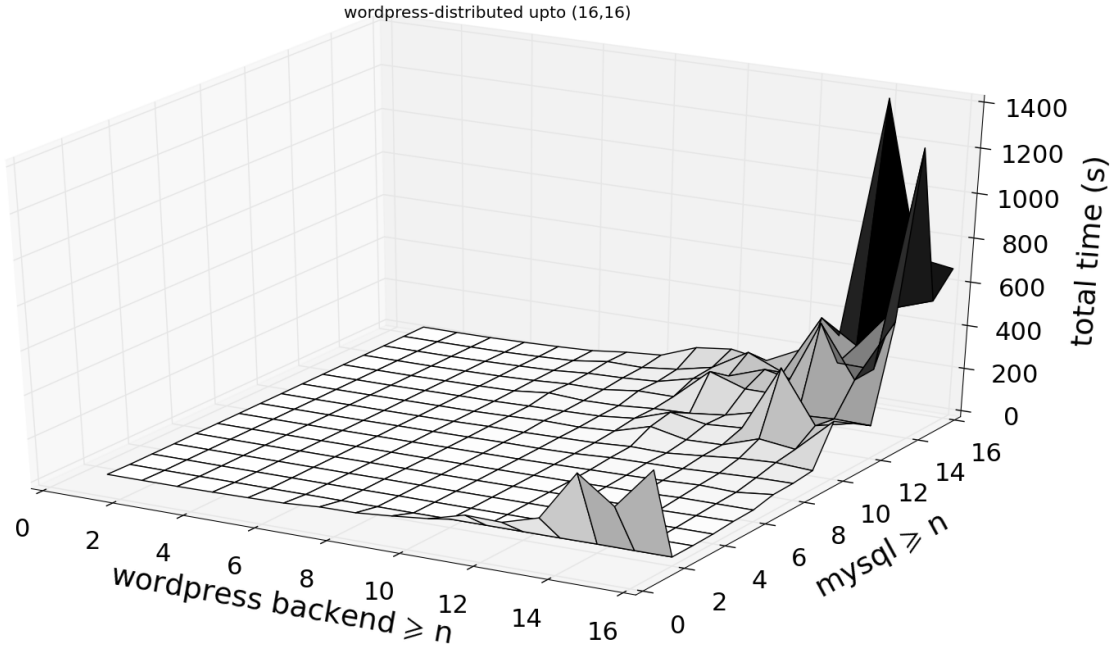


Figure 8.18: Portfolio up to (16,16).

Interpretation of the results Graphs corresponding to benchmarks solved using the G12/CPX solver show its unpredictable and chaotic nature. Although the most difficult cases correspond roughly to the highest execution times, the rest of the surface is covered with irregular ups and downs which seem to follow no discernible pattern. This is particularly visible on the first graph, the one cut at (12, 12) (figure 8.13), where we have the impression that no correlation between raising parameters and execution times exists at all. On the other hand on the full graph going up to (16, 16) (figure 8.15), the chaotic time differences between the smaller cases are less noticeable, as the vertical scale is much bigger (it goes from vertical maximum equal 350 on the (12, 12) graph, to the vertical maximum equal 350 on the (16, 16) graph).

When we employ the portfolio approach, the obtained graphs are much smoother. It appears that, in the cases where G12/CPX for some reason is less efficient, one of the other solvers takes over (i.e. solves the constraint problem first) which results in more consistent and predictable final result. One of the important conclusions which we can draw from these results is that:

- as the G12/CPX solver is powerful, but quite chaotic (in many cases it is very fast and some others it is surprisingly slow),
- and as the other two solvers (Gecode and the standard G12 solver) are weaker (i.e. slower) than G12/CPX in most cases, but much more dependable,

then integrating them all in a portfolio approach is a great way to combine their strengths and cover up their weaknesses. Basically in the cases which are well suited for the G12/CPX solver we profit from its exceptional performance and in all the cases which are difficult for it we automatically fall back on the two more basic solvers.

Before finishing, let us only mention, that the "artefact" observed before in the flat WordPress benchmarks (see subsection 8.2.4) resurfaces here again. It is visible a little better on the portfolio benchmark graphs (in the one cut at(12, 12), figure 8.16, we can see that it is even higher than the predictable peak around the point (12, 12).

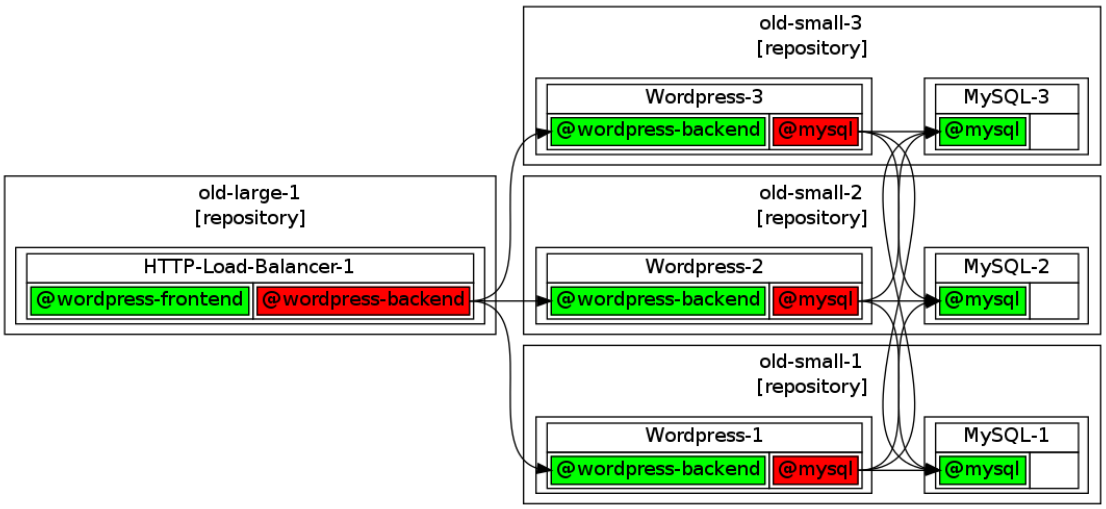


Figure 8.19: The reverse HTTP proxy based load balancer wins (as usually).

8.2.9 Optimization benchmarks

The last result which we present here is not a series of benchmarks, but just two individual scenarios which were chosen to prove a point. And the point is that Zephyrus is capable of automatically taking optimization decisions which are not obvious to humans.

We present the final configurations generated for two versions of the *wordpress-dist* scenario, which are in between two rather different solutions for the problem. All the benchmark parameters have the default values, except one: the *DNS-consume* parameter shifts from 1006 in the first case to 1007 in the second case. The exact amount of the `ram` resource that our load balancing components consume in both situations is presented in the table below:

Component Type	`ram` resource consumed	
	first scenario	second scenario
DNS Load Balancer	1006	1007
HTTP Load Balancer	8048	8056

The final configurations generated by Zephyrus in for these two scenarios are completely different:

- The first scenario (with the *DNS-consume* parameter equal 1006) gives us almost exactly what we would expect. We can see in figure 8.19, we have as usually an instance of the HTTP Load Balancer component type, plus 3 WordPress instances and 3 MySQL instances.

  What is interesting here is the choice of the locations to use and the way that the components are co-located. There are three Small machines in our configuration and each of them contains one WordPress and one MySQL. Then we have the HTTP Load Balancer placed on his own on a separate Large machine.

  The reason for such placement of components becomes clear when we compare the resources that our HTTP Load Balancer consumes (8048 units of `ram` resource) with those provided by the available machines (a Large machine provides 8052 units of `ram` resource): in fact it barely fits on its location. This is why it is not co-located with anything else.

  The total cost of the used machines = $3 \times 65 + 260 = 455$.

- The second scenario (with the *DNS-consume* parameter equal 1007) gives us a completely different outcome. As we can see in figure 8.20, this time we have an instance of the DNS Load Balancer component type, plus 7 WordPress instances and 7 MySQL instances.

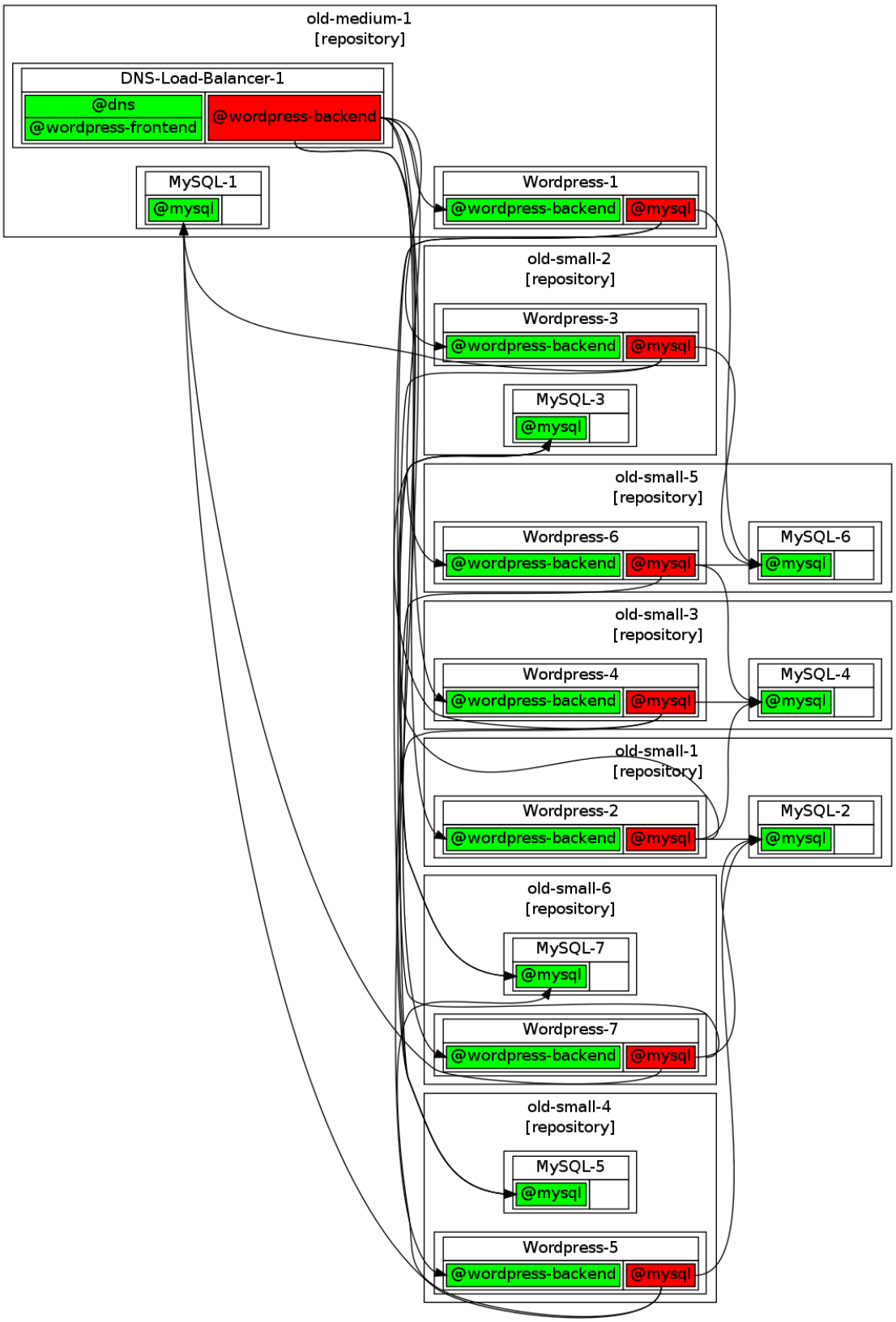


Figure 8.20: When the reverse HTTP proxy based load balancer does not fit on a Large machine it is the DNS based load balancer that wins.

There are 6 Small machines in our configuration, each with one WordPress and one MySQL, and there is one Medium machine, with both a WordPress, a MySQL and the DNS Load Balancer put together.

The reason for this sharp change is simple: in the second scenario our HTTP Load Balancer would not fit any more on a Large machine (it consumes now 8056). Therefore it would require an Xlarge machine, which is twice more expensive. In this case adding several more Small machines, in order to support all the necessary additional WordPress and MySQL instances, becomes more advantageous than paying for one Xlarge machine.

The total cost of the used machines is $6 \times 65 + 130 = 520$. On the other hand, the cost of an alternative solution similar to the first scenario (where one Xlarge machine and 2 Small machines would be necessary) would amount to $2 \times 65 + 520 = 650$.

The breaking point between 1006 and 1007 was found through experimentation, by bisecting manually a certain range of the *DNS-consume* parameter values.

8.2.10 Conclusion

In this section we have presented several benchmarks performed on the Zephyrus tool. Basically all these benchmarks were helpful in some way, they provided us with an insight about Zephyrus capacities and helped us advance. General conclusion is rather optimistic, as our tool seems to handle well reasonably complex tests of quite big size.

The main issue related with these benchmarks is that they are all based mostly on variants of a single use case. And although it is not only a very relevant and suitable, but also a widely popular use case, employed by a lot of other competing tools to showcase their capabilities, we would still like to try Zephyrus on more different realistic scenarios.

Unfortunately we have learned that good use cases which would interest us are rather hard to get by. Thus, as enterprises generally do not want to share too much information about their internal deployments for research purposes, we are obliged to stick to the few realistic scenarios that we know reasonably well.

8.3 Integration in the Aeolus tool-chain

Zephyrus is a part of the complete tool-chain developed in the Aeolus project, which is supposed to support system architects and administrators all the way from the distributed system design phase to deployment on a real architecture. In this section we discuss very briefly this tool-chain and its elements.

Components of the tool-chain

As we already know, Zephyrus is capable of automatically generating a fully detailed description of a configuration corresponding to a correctly deployed distributed system, starting from a partial abstract description of the desired system. In the Aeolus tool-chain it is thus responsible for the *system design* part. There are two more tools which constitute the tool-chain:

- **Metis**

  Metis [75] is capable of generating a full deployment plan detailing all the actions which need to be executed in order to bring the system from its current state to the state corresponding to a given configuration. Metis is thus responsible for the *reconfiguration planning* in the Aeolus tool-chain.

- **Armonic**

  Armonic [81] is a collection of tools and scripts, developed mostly by Antoine Eiche at Mandriva[2], which, starting from a knowledge base of information about available software components, allows for the deployment of software applications and services on several Linux distributions. It is responsible for orchestrating the actual *deployment* of services on machines in the distributed architecture.

Another tool, called **Blender**, realizes a software pipeline which integrates these three tools (Zephyrus, Metis and Armonic) into one seamless tool-chain and provides an interface that can be presented to the final user.

Papers

Although a paper describing the complete tool-chain is still a work in progress at the moment of writing this, another paper called *Automated synthesis and deployment of cloud applications* [24], which covers the subject of Armonic and its integration with Zephyrus, has been already published.

[2] `http://www.mandriva.com/`

8.4 Industrial validation in Kyriba

Zephyrus was deployed in a large industrial use case at Kyriba Corporation[3]. In this section we offer a return on that experience, validating the usefulness of our approach in an industrial setting. The following is based on the article *Automated Synthesis and Deployment of Cloud Applications* [24].

8.4.1 Continuous integration in Kyriba Corporation

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

Software qualification processes

Kyriba distinguishes two software qualification processes:

- a **local** one run by individual developers on their machines, detailed in figure 8.21;

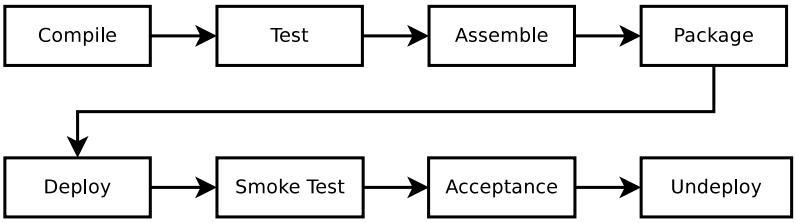


Figure 8.21: Kyriba: local qualification process.

- and a more thorough one run on a **remote** continuous integration service, depicted in figure 8.22.

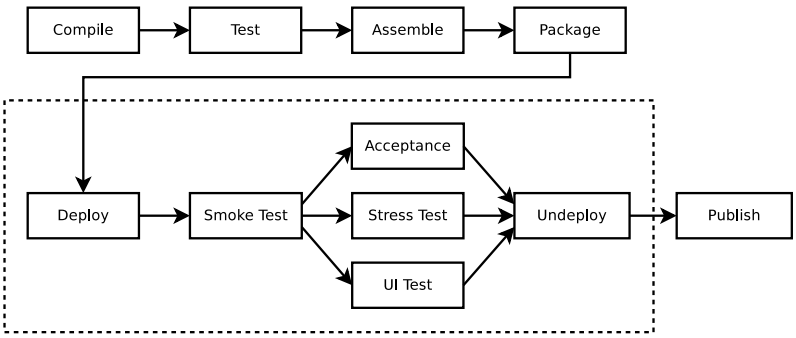


Figure 8.22: Kyriba: remote qualification process.

Heavy, exhaustive tests are performed remotely, whereas individual developers only run a subset of available tests on their machines. Successful completion of the local qualification process by a developer is required in order to be able to commit code changes to the source version control system. After each commit the remote process is triggered:

1. first the remote continuous integration service runs the same process that has been run on the developer's machine;

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

2. then automatic deployment is performed on the cloud infrastructure with the latest components version, and more extensive tests are executed: UI, deep functional scenarios, stress tests, etc.

A case for automation by Zephyrus

Kyriba follows the continuous integration recommendations [39] and implements acceptance and stress tests. These tests are very time consuming: while local tests take less than 4 minutes to complete, global ones might take 4–8 hours.

Furthermore, as *Kyriba solution* is an assembly of multiple components, integration tests involve many interdependent components that should all be deployed before testing. When deploying on a single machine, maintaining consistency (e.g. version alignment) is rather easy and can be enforced using package dependencies. However, when components are distributed as services on multiple (physical or virtual) machines, consistency is much harder to maintain.

Before, test deployment was done using custom tools involving a manual setup, and component / protocol incompatibilities were only detected at runtime. As short feedback loops usually help developers with error diagnostic related to small code changes [62], Kyriba has been looking for a tool that could anticipate error detection.

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

8.4.2 Zephyrus adoption

For developers

Zephyrus input files are available to all developers, who are in charge of maintaining the Zephyrus metadata for the components they are working on. Relationships between components are defined by developers when they create packages for their software. Two kinds of such relationships need to be defined:

- Service binding relationship with API level requirement.

  For example: the application 1.0 requires a service API version 1 exposed by another application on some machine, not necessarily where the application is deployed.

- Dependencies between packages with specific version requirements.

  For example: the application 1.0 requires a web server of version at least 3.2.4 to be installed on the same machine to run properly.

Universe Developers can declare these requirements using ports specified in the Zephyrus universe definition and the implementation relation:

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

In this example:

- The component type `kyriba-fa-accounting-engine` provides a `fa-accounting-engine-v1` service with API level 1 and requires a `graphite-v3` service.
- In the implementation section, the `component_type` is linked to the concrete package implementation `kyriba-fa-accounting-engine (= 0.1)`.

This partial universe definition is merged with a full universe description (containing the definitions of all Kyriba components) and the default specification in order to verify that at least one final configuration exists. This ensures that no dependency problem can arise during deployment.

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

Static validation The local qualification process is modified by adding a Zephyrus validation stage before (local) deployment, see figure 8.23. Using Zephyrus meta-data developers simply declare component interfaces and the way they are exposed. Moreover, using Zephyrus, developers know beforehand if the components they are working on can be deployed together with other components.

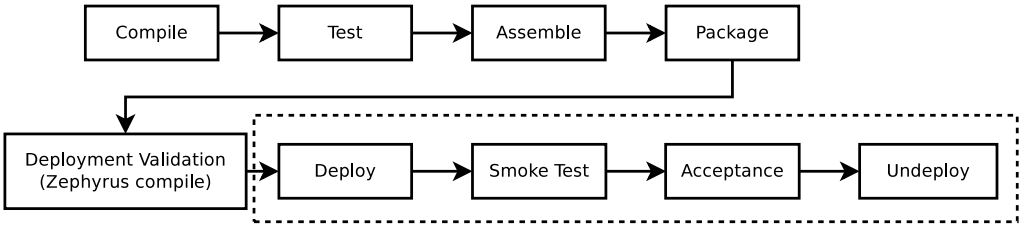


Figure 8.23: Kyriba: local qualification process with Zephyrus

For continuous integration and deployment environment

Similarly to the local qualification process, the global one has been extended with an extra Zephyrus validation stage as illustrated in figure 8.24.

Zephyrus is capable of automatically checking the application's consistency without actually engaging in the deployment process. Then, if the solver finds a solution, the produced final configuration can be used by infrastructure management scripts based on the Fabric library[4] in order to orchestrate an integration test deployment on the cloud infrastructure (such as the Amazon Elastic Cloud Computing service). The newly created platform is then tested against all acceptance, stress and UI validation scenarios.

To upgrade the production platform

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

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

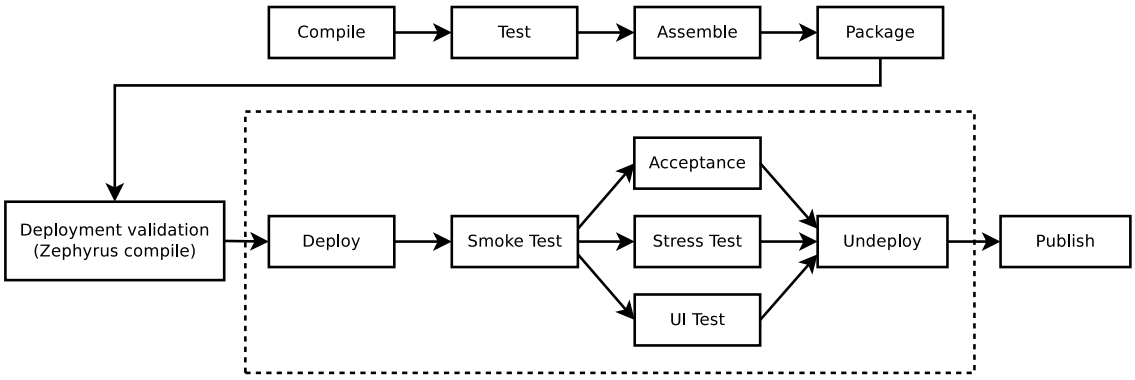


Figure 8.24: Kyriba: global qualification process with Zephyrus

8.4.3 Conclusion

Summing up, Kyriba's experience with Zephyrus is positive:

- Before Zephyrus adoption, Kyriba was managing deployment scenarios manually, using spreadsheets and flat documents, with ad-hoc semantics leading to complex, time-consuming and error-prone deployments. Zephyrus has brought precise semantics and simplifies the automation of software qualification processes.
- Zephyrus also helped by providing possibility of static validation before performing dynamic validation at runtime, which is quite expensive and very long. Zephyrus improved engineering quality and reduced building cost by reducing the number of failures both at deployment, integration test and platform upgrade stages.

Unfortunately, because of the corporate environment context, we do not have access to all the details of the use cases described in this section, and we report only the information that have been made publicly available.

8.5 Conclusion

In this chapter we have presented our validation effort of the usefulness of the Zephyrus tool, both as a standalone tool and in the context of the whole Aeolus approach. The evidence provided suggests strongly that Zephyrus is a valuable piece of software, capable of treating with a sufficient efficiency many interesting architecture synthesis problem instances, which can be used with success in an industrial environment. Together with the other components of the Aeolus tool-chain, it can be employed to perform software deployments in real-world distributed systems.

CHAPTER 9

RELATED WORK

In this chapter we attempt to enumerate and discuss briefly all the work that we know of which is related in a significant way to various elements of the work described in this thesis.

The subject that we have tackled is definitely not a new one, but we believe our approach to this subject to be innovative and different from all the other ones mentioned here. In the following we will try to show both the points of contact and the most important differences between various parts of the Aeolus approach and the ideas or methods which can be found elsewhere.

9.1 Aeolus formal model

To the best of our knowledge, Aeolus was the first model designed on purpose to formally address the specific needs of software component deployment in the cloud. Let us compare the approach we have adopted to related formal models considered in slightly different contexts.

9.1.1 Black-box / grey-box with an interface

The idea to specify a component by means of a black-box with an interface that exhibits its behaviour to the (outside) environment is widely adopted. For instance, the standard definition of component in the UML specification [95] sees components as black-boxes that may provide and require certain interfaces.

This is sometimes not enough and the inner structure of a component must be also considered. In the Aeolus model, components are in fact not black-boxes, but grey-boxes. They are fitted with provide and require ports and with an associated automaton, that describes the component's life-cycle by expressing for each internal state the corresponding ports which are active or not in that state.

9.1.2 Interface automata

Automata have been adopted long ago in the context of component-oriented development frameworks. One of the most influential model are *interface automata* [28], where automata are used to represent the component behaviour in terms of input, output, and internal actions. Interface automata support automatic compatibility check and refinement verification: a component refines another if its interface has weaker input assumptions and stronger output guarantees. The concept of interface automata have been further enhanced by Larsen et al. [74] who proposed to use two distinct automata: one for the specification of the assumptions on the environment and one for the description of the guarantees afforded by the component.

These models allow to develop formal verification methods for properties of interest and they focus on checking component compatibility and behaviour refinement. Differently from that approach, we are not interested in component compatibility nor refinement, and we do not require complementary behaviour of

components. We simply check, in a certain configuration, whether all required functionalities are provided by currently deployed components. The automata in the Aeolus model do not really represent the internal behaviour of components, but rather the effects of external deployment or reconfiguration actions on the component.

9.1.3 Petri nets

Aeolus reconfiguration actions show interesting similarities with transitions in Petri nets [98], a very popular model born from the attempt to extend automata with concurrency. At first sight one might encode the Aeolus model in Petri nets by representing:

- Aeolus component states as places in a Petri net,
- each deployed Aeolus component as a token in the corresponding place in the Petri net,
- and Aeolus reconfiguration actions as transitions that cancel and produce tokens in that Petri net.

Achievability in Aeolus would then correspond to *coverability* in Petri nets.

However, there are several important differences between the actual Aeolus model and its encoding in Petri nets. Multiple state change actions can atomically change the state of an unbounded number of components, while in a Petri net each transition consumes a predefined number of tokens. And, more importantly, we have proved that achievability can be solved in polynomial time for Aeolus$^-$ and that it is undecidable for the full Aeolus model, while in Petri nets coverability is an ExpSpace problem [103].

9.1.4 Pi-calculus

Several process calculi extend or modify the π-calculus [86] in order to deal with software components. The Piccola calculus [6] extends the asynchronous π-calculus [86] with *forms*, first-class extensible namespaces, useful to model component interfaces and bindings. Calculi like KELL [104] and HOMER [16] extend the core π-calculus with hierarchical locations, local actions, higher-order communication, programmable membranes, and dynamic binding. More recently, MECo [88] has extended this approach by proposing also explicit component interfaces and channels to realize tunnelling effects traversing the hierarchical location boundaries.

On the one hand, all these proposals differ from the Aeolus model in that they focus on the modelling of component interactions and communication, while we focus on their interdependencies during system deployment and reconfiguration. On the other hand, we could potentially take inspiration from these calculi in order to extend our model in the future with boundaries and administrative domains.

9.1.5 Fractal model

The Fractal component model [15] (which in fact pre-dates the popularization of the "cloud computing" expression) focuses on expressivity and flexibility: it provides a general notion of component assembly that can be used to describe concisely, and independently of the programming language, complex software systems.

Fractal provides an object-oriented API to manage the life-cycle of components which, in spirit, is close to what Aeolus model aims to do with component automata. However, this object-oriented API approach is limited when it comes to the ability to reason on component activation. In Aeolus we can, within limits due to the problem complexity, reason on component activation and automate it, whereas in Fractal an external reasoner will have to stop at the API invocation level, without knowledge of what a specific method implementation will do.

Also, in the Aeolus model component states are not limited to *active* or *inactive*, each component type can define its own life-cycle in detail.

9.1.6 SmartFrog model

A declarative approach, similar to Aeolus model, for modelling individual components of a system, together with their possible configuration states, is also introduced in the position paper [55]. However, the lack of a formal semantics for the approach makes it impossible to analyse the complexity of the deployment problem in this setting. Moreover, as observed by the authors, their approach allows administrators to write erroneous models presenting deadlocks or livelocks that are difficult to detect and forbid reaching the desired target configuration.

9.1.7 UML validity

To check validity of UML specifications with multiplicity constraints that can include both upper and lower bounds, Feinerer and Salzer [45] propose an encoding using linear inequations and present a polynomial time algorithm to find a configuration, and an exponential algorithm to build minimal configuration models that also involves a binding generation algorithm. This work has strong formal similarities with ours, but different motivations, which lead to incomparable models. More recent work on UML class diagrams [44] accommodates a notion of sub-classes, which amounts to a restricted form of disjunction, but even with this extension the model remains incomparable with ours.

Along similar lines, ARCAS [70] automatically synthesizes connector architectures [84] thanks to the Boolean solver embedded in the Alloy Analyzer. The main difference is in the expressivity of the component models: ARCAS specifications do not have capacity constraints, which greatly simplifies architecture and binding generation.

9.1.8 Reconfigurations

To the best of our knowledge, there is no work that formally studies the complexity of automatic reconfiguration of component systems.

A significant part of the related literature (see [111, 22, 54, 119]) focuses on the problem of dynamic re-allocation of resources. Other works focus on the nature of the reconfiguration problem, like in [121] where a classification of the reconfiguration problems is made based on its causes, namely failures, system updates, and user requests. This work, however, does not consider the complexity of establishing the reconfiguration steps.

Formal methods have been used to study reconfiguration problems as, for instance, in [73] where graph transformations and model checking are used to reason about dynamically changing component connectors. The focus in this case, however, is on proving properties of the reconfigured system.

9.2 Tools and methods for mastering cloud complexity

Let us summarize the technologies and tools that we know about, coming from both industry and academia, which are used in the context of designing, deploying and managing large distributed systems. Taking as reference the Aeolus tool-chain, we will discern roughly three main challenges which such technologies attempt to address:

- **System design**: automatizing the selection / distribution / interconnection of the software components in order to design a desired target system configuration;
- **Reconfiguration planning**: generating a sequence of actions which should be performed in order to deploy a certain system;
- **Distributed software deployment / management**: deploying and managing software in an actual real-world distributed system environment (e.g. a cloud).

Some of the existing tools concentrate on a single particular aspect of these challenges, while others try to tackle all of them simultaneously in a unified manner.

9.2.1 Managing networks of interconnected machines

The problem of finding a suitable technique to automatize the deployment of complex systems assembled from a large number of interconnected components has attracted significant attention in the area of system administration.

Different languages with their deployment engines have been proposed [43, 56, 93], but have seen limited practical adoption thus far. The most common solution for the deployment of a cloud application is still to rely on pre-configured virtual machines [48, 21, 8].

A popular, but more knowledge-intensive solution, is to use configuration management tools (already discussed in chapter 3), such as CFEngine [17], Puppet [100], and Chef [96], which significantly ease the deployment of software. Despite their differences, such tools basically allow to declare the components that should be installed on each machine, together with their configuration files. Then they employ various mechanisms to deploy these components accordingly. The burden of specifying where components should be deployed, and how to interconnect them is left to the system administrator, let alone the difficult problem of optimal resource allocation. Thus, distributed configuration management tools allow to automatize the process of carrying out the deployment of components on a pool of machines, provided a deployment plan is known beforehand.

As an additional complication, such tools stop at the package management abstraction level and rely blindly on existing package managers. Hence they have no way of knowing *in advance* whether deployment will succeed or not. For example, if the system administrator requests to install two incompatible web servers on the same machine, the incompatibility will only be discovered by the package manager at deploy time, when one of the two services fails to get installed or started. At that point, it is up to the system administrator to go back to the planning stage and work around the incompatibility.

CloudFoundry [120] is yet another tool of such kind, which specifically targets application deployment in the cloud, but suffers from the same limitations as described above.

From the Aeolus tool-chain perspective, all of these distributed configuration management tools can be potentially used as deployment back-ends, i.e. alternatives to Armonic. Once optimal resource allocation is done (by Zephyrus) and the reconfiguration plan is prepared (by Metis), the actual deployment can be delegated to them, now with the guarantee that no deployment error due to incompatibilities will arise. An interesting candidate for such use could be the distributed extension of the Nix package management system [117].

Another approach to automated deployment is proposed in [41], which uses an Architecture Description Language with user-provided information about relationships among software services, and implements a decentralized protocol to perform automatic configuration. This work might also be used as a deployment backend in the Aeolus tool-chain.

9.2.2 Computing the (optimal) final configuration

Different tools aiming to compute the (optimal) desired target configuration of a distributed system, without computing the deployment steps needed to reach it, exist.

In [85] a prediction based online-approach is proposed to find optimal reconfiguration policies, in [52] a genetic based algorithm is used to support the migration and deployment of enterprise software with their reconfiguration policies, in [115] integer linear programming methods are used to find energy efficient optimal final configurations.

However, none of the tools that we are aware of allows to declare capacity or replication constraints, which are essential non functional constraints for any non-trivial, scalable application. Furthermore, most of them give no guarantee of optimality like minimizing the number of needed (virtual) machines.

9.2.3 Other tools related in one or more aspects

ConfSolve

In ConfSolve [61], like in Zephyrus, a constraint based approach is used to propose an optimal allocation of virtual machines to servers, and of applications to virtual machines. An object-oriented declarative language is used to describe the entities (e.g. machines and services), the constraints, and the optimisation

criteria. A final configuration is then computed by translating the declarative specification into a constraint optimisation problem which is then solved by using a constraint solver. However, ConfSolve does not handle connections among services, nor capacity or replication constraints, and is unaware of package incompatibilities.

Engage

Engage [47] is very close to Aeolus purposes. It provides a declarative language to define the system configurations and a deployment engine. Engage uses automata to specify a component's behaviour and it is able to deploy the resources completing a target partial configuration, relying on a constraint solver to plan deployments. However, Engage offers no support for conflicts in the specification language. One can only indicate that a service can be realized by exactly one out of a list of components. Also, when generating the right order in which deployment actions should be performed, it relies on the assumption that the dependency graph is acyclic, meaning that circular dependencies among components are not admitted (in fact in order to produce a deployment plan it simply performs a topological sort on the graph representing the component dependencies).

Juju

Juju [19], supported by Canonical[1], includes a GUI that allows the application manager to design a final configuration and indicates the steps needed to obtain it. It cannot however compute the final and optimal configuration starting from a partial specification, nor devise an optimal order in which the deployment actions need to be performed.

In Juju, each service is deployed on a single machine (or, more recently, in a virtual container on a machine). That avoids the issue of component incompatibilities and thus the need of dealing with potential conflicts, but does so at the price of wasting resources. Moreover, Juju does not support replication constraints, and gives no guarantee of deployment optimality.

Saloon

Saloon [101] computes a final configuration by describing a cloud application using a feature model extended with feature cardinalities. It automatically detects inconsistencies, but like any other tool that we are aware of, it does not offer the ability to minimize the number of resources and virtual machines to be used.

Fractal and FraSCAti

Building on the already mentioned Fractal [15] component model, which defines a concise and expressive way to describe of a complex software system, the FraSCAti [106] platform provides middleware that can be used to deploy applications in the cloud. However, it is up to the system designer to select the components and to realize their interconnections.

VAMP [42] introduces extensions to architectural description languages to enable self-configuration, which can be seen as a dynamic approach to architecture synthesis. However, it has no notion of specification and the bindings between services are predefined.

Off-the-shelf planners

A relevant research direction concerning reconfiguration planning is to leverage traditional planning techniques and tools coming from the artificial intelligence domain. In [60, 59, 10] off-the-shelf planning solvers are exploited to automatically generate (re-)configuration actions. However, in order to use these tools all the deployment actions with theirs preconditions and effects need to be properly specified in a formalism similar to the Planning Domain Definition Language (the *de facto* standard language for planners). This hinders the usability of these kinds of tools for system administrators.

[1] `http://www.canonical.com/`

Although the Aeolus approach does not relieve administrators from the need of expressing the dynamic behaviour of components, it allows to do so more succinctly, and in a formalism that is independent from low-level planning tools. It relies on simpler and natural descriptions, like for example state machines describing the temporal order of the component configuration actions.

Bootstrapping

Also closely related to our work is [5] which proposes an heuristic-based algorithm to remove build dependency cycles for bootstrapping a Linux software distribution. The building order of the packages is generated using a topological sort of a graph.

However, the problem addressed there is slightly but significantly different from ours as the aim is to build packages from scratch. Differently from our work, one of the assumptions is that once a package is updated, its old versions are no longer required, as at every iteration the newer version of the recompiled package replaces the old one (a package is gradually built adding more and more functionalities). In our case this is not always true as the same service may be required in both `installed` and `running` state, for example in order to support fault tolerance via replication.

Service oriented architectures

In the field of service oriented architectures several approaches have been proposed to automate service *composition* and *reconfiguration*, which are both major use cases for our work. Dynamic aspects of service reconfigurations are captured using specialized architecture description languages in [46], whereas various approaches to automatically realize a given dynamic service behaviour have been studied, e.g., in [18, 12]. Our work is complementary to these studies, as it can be used to identify a static architecture of services that can then be composed as needed.

CHAPTER 10

CONCLUSION

In order to conclude this document, in the last chapter we attempt to summarize and recapitulate in a synthetic way all the work that we have accomplished and presented here. We try to put each part of it in the context of our initial aims and pinpoint the potential areas of future improvements.

While doing this we follow broadly the order in which things were introduced before: we begin with the Aeolus and Zephyrus formal models, then we discuss our constraint-based approach to solving the architecture synthesis problem (seen from the abstract formal perspective), and finally we move on to more concrete and practical contributions related directly to the Zephyrus tool.

In this document we have already mentioned several promising directions for further study and experimentation. Here we regroup the most appealing among these ideas in a more systematic fashion and add some new ones.

10.1 Aeolus formal model

The Aeolus component model, which we have introduced in detail in chapter 4, is the formal base of most of our work. In section 4.5 we have presented rapidly two restricted variations of this model: Aeolus core and Aeolus$^-$. Then, in chapter 5, the stateless version of the Aeolus model was defined and used to establish our constraint-based approach to architecture synthesis. Afterwards, this model version was further extended in chapter 6 and took form of Zephyrus model, richer and more adapted to direct practical applications.

10.1.1 Contributions

The Aeolus model was designed while keeping in mind many different models appearing in the existing approaches (enumerated in chapter 9), especially the explicit or implicit ones used in the distributed configuration management tools (discussed in chapter 3).

It was established precisely on the level of abstraction appropriate for its purpose, right above the level of configuration management tools like CFEngine, Puppet or Chef. It was tailored to describe the relations between services running in a distributed system from the administrative perspective, including important non-functional properties of such relationships, like redundancy and capacity constraints, but without getting into low-level details.

By comparing the Aeolus model with all the other models of similar scope and purpose known to us, we clearly see that it different from each of them, and it is the only one which attempts to express in the same time the internal state of software components, the dependencies and conflicts between them, as well as the non-functional requirements: redundancy and capacity. Also, it seems that the abstraction level of the model was chosen correctly, as Armonic, the deployment tool from the Aeolus tool-chain (see section 8.3),

is capable of refining the elements of the model and actually deploying the described software in a real distributed system. Thus it is certainly not a purely academic model, detached from reality and with no practical value.

The limiting theoretical results concerning the reconfigurability and achievability problems (presented in section 4.5), based on the foundation of the Aeolus model, give us an important indication about the potentially enormous complexity of the challenges related with Cloud reconfiguration.

10.1.2 Future work

Several opportunities of potential improvement present themselves in the context of different variants of the theoretical Aeolus model.

Model restrictions

First, more interesting restrictions on the Aeolus model (i.e. models between Aeolus$^-$ and the full Aeolus model) could be considered, in order to investigate the possible different trade-offs between expressivity and complexity. This kind of exploration could possibly lead to a better understanding of which inherent aspects of distributed systems cause most difficulty in the challenges related to automation and reconfiguration.

Hierarchic extension

Second, the full Aeolus model could be extended to a hierarchical component model, incorporating such concepts as administrative domains, components built by grouping together other components and nested virtualization containers. This would permit us to greatly improve its flexibility and expressivity, which would be particularly appreciable in practical applications.

There are many reasons why this is a very difficult task. Mainly, when tweaking our theoretical model we must take great care in assuring that we are able to adapt our whole approach to the model-level changes which we introduce. Notably, if we cannot encode all the features of our model in constraint form, it will not be possible to use the new version of the model in practice. Unfortunately, for now all the attempts to design a satisfying hierarchic extension of Aeolus model which works well with our approach have failed.

Reintegrate the Zephyrus model features

It is also worth mentioning, that if we actually managed to develop this kind of a hierarchic extension of Aeolus model, then a new interesting possibility would open: we could use the hierarchical structure in order to reintegrate the Zephyrus model features back into the stateful Aeolus model (thus creating a stateful Zephyrus model). In such a hierarchic model conflicts could be made local to a given subspace and thus some entities, like packages, could be modelled differently, potentially in a more uniform and elegant way.

10.2 Zephyrus model

Zephyrus model is an extended variant of the stateless Aeolus model. It was introduced and described in detail in chapter 6. Then, as it is employed directly in the Zephyrus tool, the model and its different parts are examined in various contexts on many occasions in chapter 7, when discussing its practical representation and the related implementation issues, as well as in chapter 8, when assessing the Zephyrus utility and efficiency.

10.2.1 Contributions

The main purpose behind the Zephyrus model was to adapt the stateless Aeolus model to practical considerations. This basically amounts to incorporating the problem of component placement into the model by extending it with the concept of locations and adding all the pieces of information (i.e. package repositories

and computing resources) necessary to take informed decisions concerning placement of components on available locations.

Zephyrus model is pretty successful in realizing its intended purpose. It is slightly less elegant than the pure Aeolus model (stateless or stateful), as it contains concepts considered as relatively low-level (i.e. repositories, packages and computing resources, which should rather belong to the deployment layer of abstraction, though need to be included here to guarantee correct component placement), but these concepts are introduced in a way that is still quite abstract and they integrate well with the base model as well as with our constraint generation method. Also, thanks to some tweaks on the practical level (like the package and location trimming techniques, described in section 7.5), the additional complexity impact from adding all the new model elements is considerably mitigated.

10.2.2 Future work

As mentioned multiple times in chapter 6, several decisions made when designing the Zephyrus model were taken at least in part arbitrarily. However, the model is not set in stone: these decisions can be always revisited if we become able to make a more informed choice and capable of implementing it in practice. Tweaking these features could possibly extend greatly the model's flexibility while not adding too much implementation complexity.

Fixed repositories

One of the topics which we have already recognised as requiring some more insight is the question of fixed or changeable repositories on locations. For now we have decided that all the repositories attributed to locations in the initial configuration can be changed in the final configuration. A simple but equally unsatisfying alternative would be to do the contrary: fix all the attributed repositories.

In order to improve on these ideas a mechanism of finer control over this aspect of configuration should be established, permitting to reflect the actual degree of administrative control that we have (or we want to have) over the machine represented by a particular location. Several concepts related to this problem have been discussed throughout chapter 6 (see, for example, section 6.4.2).

Hierarchical repositories

Another interesting idea concerning the package repositories is the question of the nature of their relationship with locations. In our current version of Zephyrus model one and exactly one repository must be assigned to each location in the configuration. In this interpretation repositories represent mutually exclusive alternatives: different operating systems which can be installed on a single machine.

We could imagine another interpretation: repositories representing actual package sources, where several of them can be used simultaneously on a single machine. In this case we would permit a set of repositories to be installed on each location and we would introduce dependencies and conflicts between repositories. In fact our repositories could be modelled almost exactly as our packages are modelled now.

Virtual components

Yet another concept which was already mentioned in chapter 6 is related to attributing components to locations. In the current Zephyrus model each component needs to be placed on one and exactly one location. This comes directly from the part of Aeolus philosophy concerning components, established in the preceding chapters (especially see chapter 4), that every component represents a service deployed on a single machine.

In practical situations we might be potentially interested in using some *virtual* components in our configurations: components which are not installed on any particular machine. These might be used to represent some external services available in the environment, etc.

Specification expressivity

The specification syntax used in the extended architecture synthesis problem is very expressive. However, as it was not created by looking from the system designer point of view, but rather by based on "what can we easily encode in form of constraints" philosophy, it is in fact not very well adapted to expressing many typical user requests and deployment policies.

For example, if we want some condition to be true on every location, we encode it through *counting* all the locations which *do not satisfy* this condition and declaring that their number must be *equal to zero*. This is not a very direct nor efficient constraint encoding of a simple request. We could extend our specification syntax with some new constructions, like $everywhere(S_l)$ (on every location the local specification S_l must be satisfied), which would provide direct and natural encodings of common properties.

Moreover, our full specification syntax as it is defined now is actually *too* expressive in many aspects (e.g. arbitrary use of free variables gives it considerable generic expressive power). This degree of expressivity limits greatly our capacity to reason over the extended architecture synthesis problem, while in the same time it does not provide much actual utility. Tweaking the specification syntax, so that it attains the required level of expressivity, could help us thus not only in making our approach more flexible, useful and efficient in practical applications, but also by reducing its internal complexity.

10.3 Constraint-based approach

The architecture synthesis problem (based on the stateless Aeolus model) and the constraint-based approach to solve it were introduced in chapter 5. Subsequently, the extended version of the problem (based on the Zephyrus model) as well as the corresponding extended version of the constraint-based approach were discussed in chapter 6. Then, in the subsequent chapters, 7 and 8, we have seen how the approach was implemented in practice in form of the Zephyrus tool, and how its pertinence and efficiency was assessed (indirectly, i.e. through this implementation) in several different contexts.

10.3.1 Contributions

The idea of using finite domain constraint satisfaction and optimization to solve problems related with configuration synthesis and deployment planning is definitely not new. Two good examples of tools which rely on such techniques are ConfSolve [61] and Engage [47], mentioned in chapter 9.

In our case applying this method was quite challenging though, mostly because of the non-functional requirements (i.e. redundancy and capacity), which are an important feature of the Aeolus and Zephyrus models. These were not exactly easy to encode in the integer constraint form. The way this issue was surmounted is based on an ingenious combination of two ideas: taking advantage of the *binding unicity* property, which is inherent to our models, and generating the bindings between components *after* the constraint resolution phase, using so-called *matching algorithm*. Both these ideas were discussed in section 5.3.

10.3.2 Future work

Although many practical enhancements of the efficiency our constraint-based approach have been proposed and implemented on the Zephyrus tool level, there is also some room for improvement on the theoretical level.

Eliminating multiplications from the encoding

Multiplications (i.e. more exactly: situations when two variables are multiplied) are probably the hardest parts of the constraint problems generated by our approach, from the constraint solver point of view. In our constraint problems such multiplications appear in the constraints corresponding to the binding unicity principle rules (see section 5.3.3), which are crucial to correctly encoding the final configuration's validity.

Many efforts have been made to find a constraint encoding which would be equivalent to our current one, but without the need of using multiplications. This would make our constraints linear and thus,

theoretically, much easier to solve. Moreover, a whole new category of very efficient solvers would become available to us: the MILP[1] solvers.

Unfortunately, it seems that the aim of eliminating these multiplications from our constraints in a satisfying fashion (i.e. without drastically blowing up the size of the entire encoding) is very hard to achieve or even impossible. It would be great to either prove that it is indeed impossible or find a way to do it.

Alternative binding generation method

Our matching algorithm (introduced in section 5.3.4) used to generate bindings between the components in our final configuration is efficient, but it does not allow us to control the process of binding generation at all. And in certain cases we may want to have some degree of control over the generated bindings, for example in order to specify that local bindings (i.e. bindings between components placed on the same location) should be preferred to bindings traversing the location boundaries.

It is possible to use a more flexible solver-based approach instead of a fixed algorithm. Some work has been already performed in this direction (see [82]), but it is not yet incorporated into Zephyrus at the moment of this writing.

10.4 The Zephyrus tool

The Zephyrus tool is a practical implementation of our constraint-based approach to solving instances of the extended architecture synthesis problem. Its whole theoretical foundation was presented in chapter 6 (which in turn relies highly on concepts and methods introduced in the previous chapters) and all the practical aspects of various parts of its implementation, architecture, and usage were discussed in chapter 7. We have examined it closely both from the *efficacy* angle (i.e. to understand how does it process its inputs in order to produce the right outputs), in sections 7.2 and 7.4, and from the *efficiency* point of view (i.e. how do we improve its performances in practice), in sections 7.3 and 7.5. Then the practical utility of Zephyrus and its performances in different scenarios were investigated in chapter 8.

10.4.1 Contributions

Zephyrus is a research prototype tool, written in OCaml [79] and using many other existing technologies (from basic ones like the JSON language [26], to more complex ones such as the coinst tool [25] or ATDgen [68]). We can see Zephyrus from two fundamental perspectives: as an implementation of our theoretical constraint-based approach and as a real-world distributed system architecture synthesis tool.

Looking from the first perspective, we should underline that it attempts to realize the formal approach in practice while staying as faithful to it as possible. Thanks to this effort, we can be quite confident that all the advantageous properties of our approach, which was formally proved to be correct, are preserved on the implementation level (of course if we do not account for potential bugs in the code). Thus Zephyrus can serve as a proof of concept for our approach: it is a validation of the theoretical idea.

From the other side, Zephyrus is an actual piece of useful software which works in the real-world and which can be employed to assist in distributed system design and deployment. The value of Zephyrus in this context has been established in two ways: first by its integration in the Aeolus tool-chain, which is capable of using the optimal configuration descriptions generated by Zephyrus in order to deploy real software on some virtual machines (as briefly described in section 8.3); and second by the fact that Zephyrus was successfully used in a corporate industrial environment (at Kyriba Corporation[2]) in order improve the efficiency of the continuous integration process and the deployment of test-environments (as presented in section 8.4).

[1] Mixed Integer Linear Programming

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

10.4.2 Future work

Zephyrus still requires a lot of work in order to be considered mature and it offers multiple potential angles of improvement. Here we list only a few of them, but we should keep in mind, that almost every aspect of Zephyrus might be at some point considered as worthy of attention and further development.

Custom optimization functions

Zephyrus offers currently quite a restricted repertoire of optimizations functions. As already discussed in section 7.2.3, it would not be too difficult to add a way for the user to declare custom optimization criteria. We could base them on the basic building blocks of our abstract optimization criteria syntax (see table 6.4):

- minimize a constraint expression,
- maximize a constraint expression,
- lexicographically optimize a list of criteria,

where the constraint expressions themselves could be created using the Zephyrus specification syntax (see table 6.1).

Although there are several good reasons why we did not include this feature in Zephyrus yet (mentioned in section 7.2.3), adding it might be in fact quite practical after all, maybe not for the end user, but from the experimentation point of view.

Different concrete constraint problem encodings

Our main and only concrete constraint problem representation language in Zephyrus is MiniZinc. It fits well our purposes and gives us quite a large set of compatible constraint solver back-ends to choose from. However, there are always alternatives which might provide us with other solver back-ends, potentially more adapted to the problem instances we generate and thus more efficient.

Especially if we arrived to encode our constraints in a more restricted constraint representation than the full MiniZinc (e.g. linear programming), we could access much more specialized types of solvers. One of already mentioned (see section 7.4.7) ideas related to this concept was encoding our constraints in ZIMPL language [72] and using a quasi-linear programming solver. Of course it is difficult to know beforehand if we would really gain something from this approach (as performances of constraint solvers tend to be sometimes quite unpredictable), but it would definitely be worth trying.

Benchmarks

There is also a very large amount of work left to do in the area of practical experimentation on Zephyrus. The benchmarks presented in section 8.2 cover a very limited range of potential option combinations that we could provide to Zephyrus.

Checking how all the option combinations work in practice on all the different sizes of our basic test scenarios could give us some more insight on the efficiency of our different performance optimization techniques. Hopefully it could help us answer some doubts, e.g. in which scenarios does the no-package approach (i.e. replacing the repositories of packages by direct component incompatibilities) work in practice?

10.5 Outcome

This thesis presents and discusses in detail a whole coherent chunk of work performed in the scope of the Aeolus project. We believe that the presented research and its practical results are valuable and contribute to the state of the art in the field of the distributed application synthesis and deployment automation. We hope that they will lead both to further fruitful scientific investigation and to constructive industrial applications.

LIST OF FIGURES

LIST OF TABLES

LIST OF DEFINITIONS

BIBLIOGRAPHY

[1] ISO/IEC 14977:1996 Information Technology - Syntactic Metalanguage - Extended BNF, 1996.

[2] *time(1) Linux User's Manual*, 2014.

[3] Pietro Abate, Jaap Boender, Roberto Di Cosmo, and Stefano Zacchiroli. Strong dependencies between software components. In *ESEM 2009: 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 89–99, 2009.

[4] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, October 2012.

[5] Pietro Abate and Schauer Johannes. Bootstrapping Software Distributions. In *CBSE*, 2013.

[6] Franz Achermann and Oscar Nierstrasz. A calculus for reasoning about software composition. *Theor. Comput. Sci.*, 331(2-3):367–396, 2005.

[7] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio approaches for constraint optimization problems. In Panos M. Pardalos, Mauricio G. C. Resende, Chrysafis Vogiatzis, and Jose L. Walteros, editors, *Learning and Intelligent Optimization - 8th International Conference, Lion 8, Gainesville, FL, USA, February 16-21, 2014. Revised Selected Papers*, volume 8426 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 2014.

[8] Amazon. AWS CloudFormation. `http://aws.amazon.com/cloudformation/`.

[9] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA, 2003.

[10] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3), 2007.

[11] Adnan Ashraf, Mikko Hartikainen, Usman Hassan, Keijo Heljanko, Johan Lilius, Tommi Mikkonen, Ivan Porres, Mahbubul Syeed, and Sasu Tarkoma. *Introduction to Cloud Computing Technologies*, volume 60 of *TUCS General Publication*, page 1–41. TUCS, 2013.

[12] Marco Autili, Davide Di Ruscio, Amleto Di Salle, Paola Inverardi, and Massimo Tivoli. A model-based synthesis process for choreography realizability enforcement. In *FASE*, volume 7793 of *LNCS*, pages 37–52. Springer, 2013.

[13] Arif Bilgin, John Ellson, Emden Gansner, Yifan Hu, and Stephen North. Graphviz - Graph Visualization Software. `http://www.graphviz.org/`.

[14] Pascal Brisset and Nicolas Barnier. FaCiLe: a Functional Constraint Library. http://www.recherche.enac.fr/log/facile/.

[15] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[16] Mikkel Bundgaard, Thomas T. Hildebrandt, and Jens Chr. Godskesen. A CPS encoding of name-passing in Higher-order mobile embedded resources. *Theor. Comput. Sci.*, 356(3):422–439, 2006.

[17] Mark Burgess. A Site Configuration Engine. *Computing Systems*, 8(2), 1995.

[18] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Massimo Mecella, and Fabio Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

[19] Canonical Ltd. Juju, DevOps distilled. https://juju.ubuntu.com/.

[20] Canonical Ltd., Ubuntu community. Ubuntu: The leading OS for PC, tablet, phone and cloud. http://www.ubuntu.com/.

[21] CenturyLink. Cloud Blueprints. http://www.centurylinkcloud.com/products/management/blueprints.

[22] Hyung Won Choi, Hukeun Kwak, Andrew Sohn, and Kyusik Chung. Autonomous learning for efficient resource utilization of dynamic VM migration. In *ICS*, pages 185–194. ACM, 2008.

[23] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[24] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 211–222. ACM, 2014.

[25] 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.

[26] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[27] Douglas Crockford. Introducing JSON. http://json.org/.

[28] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, 2001.

[29] Debian Project. Debian – The Universal Operating System. https://www.debian.org/.

[30] Roberto Di Cosmo and Pietro Abate. ami-thinner. https://github.com/rdicosmo/ami-thinner.

[31] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, and Jakub Zwolakowski. Optimal Provisioning in the Cloud. Technical report, Aeolus project, June 2013. http://hal.archives-ouvertes.fr/hal-00831455.

[32] Roberto Di Cosmo, Fabio Mancinelli, Jaap Boender, Jerome Vouillon, Berke Durak, Xavier Leroy, David Pinheiro, Paulo Trezentos, Mario Morgado, Tova Milo, Tal Zur, Rafael Suarez, Marc Lijour, and Ralf Treinen. Report on formal mangement of software dependencies. Technical report, EDOS, April 2006. EDOS project Deliverable 2.2, available as http://hal-univ-diderot.archives-ouvertes.fr/docs/00/69/74/68/PDF/edos-wp2d2.pdf.

[33] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Component Reconfiguration in the Presence of Conflicts. In *ICALP 2013: 40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 187–198. Springer-Verlag, 2013.

[34] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: a component model for the cloud. *Information and Computation*, 2014.

[35] Roberto Di Cosmo, Paulo Trezentos, and Stefano Zacchiroli. Package upgrades in foss distributions: Details and challenges. In *HotSWUp'08: Hot Topics in Software Upgrades*. ACM, 2008.

[36] 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.

[37] R. Droms. Dynamic Host Configuration Protocol. RFC 2131 (Draft Standard), 1997. Updated by RFCs 3396, 4361, 5494, 6842.

[38] Berke Durak and Nicolas Pouillard. The OCaml system release 4.02; Documentation and user's manual; Chapter 18: The ocamlbuild compilation manager. `http://caml.inria.fr/pub/docs/manual-ocaml/ocamlbuild.html`.

[39] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[40] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.

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

[42] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel de Palma. Self-configuration of distributed applications in the cloud. *2013 IEEE Sixth International Conference on Cloud Computing*, 0:668–675, 2011.

[43] Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel De Palma. Self-Configuration of Distributed Applications in the Cloud. In *CLOUD*, 2011.

[44] Ingo Feinerer. Efficient large-scale configuration via integer linear programming. *AI EDAM*, 27(1):37–49, 2013.

[45] Ingo Feinerer and Gernot Salzer. Consistency and minimality of UML class specifications with multiplicities and uniqueness constraints. In *TASE*, pages 411–420, 2007.

[46] José Luiz Fiadeiro and Antónia Lopes. A model for dynamic reconfiguration in service-oriented architectures. *Software and System Modeling*, 12(2):349–367, 2013.

[47] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. In *PLDI*, pages 263–274. ACM, 2012.

[48] Flexiant. Bento Boxes. `http://www.flexiant.com/2012/12/03/application-provisioning/`.

[49] Kevin Forsberg and Harold Mooz. The relationship of system engineering to the project cycle. *INCOSE International Symposium*, 1(1):57–65, 1991.

[50] Yann Régis-Gianas François Pottier. Menhir. `http://gallium.inria.fr/~fpottier/menhir/`.

[51] Inc. Free Software Foundation. GNU General Public License version 3 (GPLv3). `http://www.gnu.org/licenses/`, 2014.

[52] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *ICSE*, pages 512–521. ACM/IEEE, 2013.

[53] GitHub, Inc. GitHub - a web-based Git repository hosting service. http://github.com/.

[54] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, Guillaume Belrose, Tom Turicchi, and Alfons Kemper. An integrated approach to resource pool management: Policies, efficiency and quality metrics. In *DSN*, pages 326–335. IEEE, 2008.

[55] Patrick Goldsack, Paul Murray, Andrew Farrell, and Peter Toft. SmartFrog and Data Centre Automation. Technical report, Microsoft Research, 2008.

[56] Glauco Estacio Gonçalves, Patricia Takako Endo, Marcelo Anderson Santos, Djamel Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mångs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *CloudCom*, 2011.

[57] M. Mahmudul Hasan, Pericles Loucopoulos, and Mara Nikolaidou. Classification and qualitative analysis of non-functional requirements approaches. In Ilia Bider, Khaled Gaaloul, John Krogstie, Selmin Nurcan, Henderik Alex Proper, Rainer Schmidt, and Pnina Soffer, editors, *Enterprise, Business-Process and Information Systems Modeling - 15th International Conference, BPMDS 2014, 19th International Conference, EMMSAD 2014, Held at CAiSE 2014, Thessaloniki, Greece, June 16-17, 2014. Proceedings*, volume 175 of *Lecture Notes in Business Information Processing*, pages 348–362. Springer, 2014.

[58] Philip Hazel. Exim Internet Mailer. http://www.exim.org/.

[59] Herry Herry and Paul Anderson. Planning with Global Constraints for Computing Infrastructure Reconfiguration. In *CP4PS*, 2012.

[60] Herry Herry, Paul Anderson, and Gerhard Wickler. Automated Planning for Configuration Changes. In *LISA*, 2011.

[61] John A. Hewson, Paul Anderson, and Andrew D. Gordon. A Declarative Approach to Automated Configuration. In *LISA*, 2012.

[62] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[63] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[64] Internet Systems Consortium. BIND DNS Server. http://www.isc.org/software/bind.

[65] Ian Jackson and Christian Schwarz. Debian policy manual. http://www.debian.org/doc/debian-policy/, 2009.

[66] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[67] Martin Jambon. Adjustable Type Definitions (ATD) language. http://mjambon.com/atdgen.

[68] Martin Jambon. Atdgen tool. http://mjambon.com/atdgen.

[69] Poul-Henning Kamp, Redpill-Linpro, and Varnish Software. Varnish HTTP accelerator. https://www.varnish-cache.org/.

[70] Jaroslav Keznikl, Tomáš Bureš, František Plášil, and Petr Hnětynka. Automated resolution of connector architectures using constraint solving (ARCAS method). *Software and Systems Modeling*, pages 1–30, 2012.

[71] Jan Kneschke. lighttpd web server. http://www.lighttpd.net/.

[72] Thorsten Koch. *Rapid Mathematical Prototyping*. PhD thesis, Technische Universität Berlin, 2004.

[73] Christian Krause. *Reconfigurable Component Connectors*. PhD thesis, Universiteit Leiden, 2011.

[74] Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface input/output automata. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.

[75] Tudor Alexandru Lascu, Jacopo Mauro, and Gianluigi Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In *The 10th International Symposium on Formal Aspects of Component Software, FACS 2013*.

[76] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02; Documentation and user's manual; Chapter 12: Lexer and parser generators (ocamllex, ocamlyacc). `http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`.

[77] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02; Documentation and user's manual; Chapter 15: The documentation generator (ocamldoc). `http://caml.inria.fr/pub/docs/manual-ocaml/ocamldoc.html`.

[78] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02; Documentation and user's manual; Chapter 21: The standard library. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Arg.html`.

[79] Xavier Leroy, Damien Doligez, Jacques Garrigue, and Didier Rémy. *The Objective Caml system release 4.01; Documentation and user's manual*. INRIA, Rocquencourt, Paris, 2013.

[80] 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 Sebastian Uchitel and Steve Easterbrook, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 199–208. IEEE Computer Society, 2006.

[81] Mandriva. Armonic. `http://armonic.readthedocs.org/en/latest/index.html`.

[82] Jacopo Mauro. Aeolus utils: binding optimizer for Zephyrus. `https://github.com/aeolus-project/utils/tree/master/bindings_optimizier`.

[83] M. D. McIlroy. Mass-produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, 1968.

[84] Nikunj R. Mehta, Nenad Medvidovic, and Sandeep Phadke. Towards a taxonomy of software connectors. In *ICSE*, pages 178–187. ACM, 2000.

[85] Haibo Mi, Huaimin Wang, Gang Yin, Yangfan Zhou, Dian xi Shi, and Lin Yuan. Online Self-Reconfiguration with Performance Guarantee for Energy-Efficient Large-Scale Cloud Computing Data Centers. In *SCC*, pages 514–521. IEEE, 2010.

[86] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I/II. *Inf. Comput.*, 100(1):1–77, 1992.

[87] P. Mockapetris. *RFC 1034 Domain Names - Concepts and Facilities*. Internet Engineering Task Force, 1987.

[88] Fabrizio Montesi and Davide Sangiorgi. A Model of Evolvable Components. In *TGC*, volume 6084 of *Lecture Notes in Computer Science*, pages 153–171. Springer, 2010.

[89] National ICT Australia. The minizinc challenge. `http://www.minizinc.org/challenge.html`.

[90] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *CP*, pages 529–543, 2007.

[91] Nginx, Inc. NGINX: High Performance Load Balancer, Web Server, & Reverse Proxy. http://nginx.com/.

[92] NVIDIA Corporation. NVIDIA CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, August 2014. Version v6.5.

[93] 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.

[94] OCamlPro. OPAM - the Package Manager edited by OCamlPro. http://opam.ocamlpro.com/.

[95] OMG Unified Modeling Language TM (OMG UML), Superstructure. Technical Report formal/2011-08-06, Object Management Group, aug 2011. Chapter 8. Components.

[96] Opscode. Chef. http://www.opscode.com/chef/. Retrieved September 2013.

[97] Oracle Corporation. MySQL: The world's most popular open source database. http://www.mysql.com/.

[98] C. A. Petri. *Kommunikation mit Automaten, PhD thesis*. Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[99] PHP Group. PHP: Hypertext Preprocessor. http://php.net/.

[100] Puppetlabs. Puppet. http://puppetlabs.com/.

[101] Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien, and Goetz Botterweck. Consistency checking for the evolution of cardinality-based feature models. In *SPLC*, 2014.

[102] Clément Quinton, Romain Rouvoy, and Laurence Duchien. Leveraging feature models to configure virtual appliances. In *Proceedings of the 2Nd International Workshop on Cloud Computing Platforms*, CloudCP '12, pages 2:1–2:6, New York, NY, USA, 2012. ACM.

[103] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoret. Comp. Sci.*, 6:223–231, 1978.

[104] Alan Schmitt and Jean-Bernard Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *LNCS*, pages 146–178. Springer, 2004.

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

[106] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA applications with the FraSCAti platform. In *IEEE SCC*, pages 268–275. IEEE, 2009.

[107] Sendmail, Inc. Sendmail: a general purpose internetwork email routing facility. http://www.sendmail.com/sm/open_source/.

[108] Amazon Web Services. Amazon ec2: Pricing. http://aws.amazon.com/ec2/pricing/, 2014.

[109] INFORMS Computing Society. Mathematical Programming Glossary. http://glossary.computing.society.informs.org/ver2/mpgwiki/index.php.

[110] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663 (Informational), August 1999.

[111] Jan Stoess, Christian Lang, and Frank Bellosa. Energy Management for Hypervisor-Based Virtual Machines. In *USENIX Annual Tech. Conf.*, pages 1–14. USENIX, 2007.

[112] Peter J. Stuckey, Maria Garcia de la Banda, Michael Maher, John Slaney, Zoltan Somogyi, Mark Wallace, and Toby Walsh. The G12 project: Mapping solver independent models to efficient solutions. In *International Conference on Logic Programming (ICLP)*, volume 3668, pages 9–13, 2005.

[113] The Apache Software Foundation. The Apache HTTP Server Project. `http://httpd.apache.org/`.

[114] Linus Torvalds. Git - a free and open source distributed version control system. `http://git-scm.com/`.

[115] Phuong Nga Tran, Leonardo Casucci, and Andreas Timm-Giel. Optimal mapping of virtual networks considering reactive reconfiguration. In *CLOUDNET*, pages 35–40. IEEE, 2012.

[116] Ralf Treinen and Stefano Zacchiroli. Description of the CUDF format. *CoRR*, abs/0811.3621, 2008.

[117] Sander van der Burg, Eelco Dolstra, and Merijn de Jonge. Atomic upgrading of distributed systems. In *Hot Topics in Software Upgrades*, pages 1–5. ACM, 2008.

[118] Wietse Venema. Postfix mail server. `http://www.postfix.org/`.

[119] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *Middleware*, volume 5346 of *LNCS*, pages 243–264. Springer, 2008.

[120] VMWare. Cloud Foundry, deploy & scale your applications in seconds. `http://www.cloudfoundry.com/`. Retrieved June 2014.

[121] Shuai Wang, Fang Du, Xinming Li, Yi Li, and Xingye Han. Research on dynamic reconfiguration technology of cloud computing virtual services. In *CCIS*, pages 348 – 352. IEEE, 2011.

[122] WordPress Foundation. WordPress: Blog Tool, Publishing Platform, and CMS. `https://wordpress.org/`.

[123] World Wide Web Consortium. Extensible Markup Language (XML) 1.1 (Second Edition). `http://www.w3.org/TR/2006/REC-xml11-20060816/`.

[124] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[125] Jakub Zwolakowski. Distributed System Administration Technologies, a State of the Art. Technical report, Paris 7 ; Inria, September 2012.