Using Constraint Solvers to Find Valid Software Configurations

Clemens Danninger (1127840)

January 31, 2015

Abstract

Currently, even large-scale and complex software systems are still configured mostly by hand, which is a time-consuming and error-prone task. The automatic generation of software configurations from declarative specifications appears to be a promising solution for this problem. Constraint programming, where constraint solvers generate concrete configurations as solutions to constraint problems, has been proposed as an interesting method in recent research.

This paper gives an overview on the basics of constraint programming, and then presents various approaches and implementations to the automatic generation of valid software configurations using constraint solvers.

1 Introduction

Modern computing environments as diverse as distributed computing systems, multi-component embedded systems, and cloud computing applications (Software/Platform as a Service etc.), pose highly complicated configuration problems. A large variety of requirements, connections and resulting dependencies, which are often implicit and hidden, need to be considered.

For example, multiple services may be running on a variable number of physical machines, and might be dynamically migrated among them, this requiring adaptive assignments. Cost is an important factor, especially with usage-based billing, so it is advantageous to (automatically) optimize resource utilization. Additionally, fault tolerance (including automated repairs) needs to be considered.

Furthermore, deployment and redeployment take place frequently. Unlike when systems were usually installed just once and then left running for months or years with essentially the same setup, virtualized cloud systems often scale dynamically, i.e. servers are started and stopped according to demand.

Currently, configuration of such systems is generally done by human experts. However, it is very hard to always consider all applicable constraints and interdependencies, especially for long dependency chains, and the process is correspondingly tedious, time-consuming and error-prone.

A solution to this dilemma, which will be discussed in this paper, is to specify constraints explicitly instead, using a constraint language. Then, a constraint solver is used to find concrete values which satisfy all applicable constraints, and to thereby generate a valid configuration ready for deployment.

A system configured this way would be more reliable, since explicitly specified constraints are guaranteed not to be violated – provided, of course, that the constraints themselves are correct.

This approach is relatively new, and applications in hardware configuration actually predate those in software configuration. For example, Stumptner et al. published their COCOS system [18] in 1994, and Fleischanderl et al. [7] pursued a similar direction in 1998.

In the software domain, applications have been proposed ranging from configuration in Service Oriented Architectures [14] and cloud orchestration [13] to automotive embedded systems [16]. Previously, automatic configuration generation has appeared in expert systems used for configuration of combined hardware-software assemblies, such as Digital's XCON [1].

Like the selection of papers presented here, current research is mostly focused on large-scale component combination and assignment of components to (physical or virtual) machines, as well as on describing interfaces and interactions between components. There is comparatively little focus on conventional ”small-scale” parameter specification.

The remainder of this paper is structured as follows: Section 2 gives an overview of constraint programming and constraint satisfaction problems (CSPs), and how can it be used for configuration tasks. Section 3 contains a survey of existing approaches and implementations, and Section 4 provides a conclusion and an outlook for the concept.

2 Overview

In constraint programming, rather than actual values or methods to calculate them, only constraints over variables are specified – it is declarative, not imperative. Constraints specify which values are valid for a variable, which may also depend on other variables.

A constraint solver then attempts to find one or more assignments to the variables which satisfy all constraints. Since the solver can freely choose values as long as all constraints are satisfied, the assignments can be optimally chosen with regards to some objective function, i.e. chosen to maximize or minimize this function.

An example of a constraint program could be $x < 5$, $x \neq y$, $x + y = 6$, $x, y \geq 0$. A valid assignment would be $x = 1, y = 5$. On the other hand, $x = 3, y = 3$ would be an invalid assignment, since it violates the constraint $x \neq y$. For optimization, with the objective to maximize x, a solver would (ideally) choose $x = 4, y = 2$.

Of course, real-world applications are typically far more complicated and may involve thousands of constraints. In general, solving constraint satisfaction problems is NP-hard. For real-world problems however, it is usually possible to find an adequate solution within reasonable time. In fact, it frequently turns out that memory, not time, is the limiting factor [9, 10, 11]. General-purpose solvers such as Gecode [8], where much effort has been spent on optimization, will generally outperform custom-written solutions, in addition to saving significant effort.

In the context of configuration, constraint solving and optimization relieves the user from the often very complex task of finding specific values, and manually confirming that they are valid and compatible with each other. Furthermore, it is possible to automatically determine conflicts between constraints.

To declare constraints, most approaches (e.g. [9], [4] and [11]) use domain-specific constraint programming languages. Usually, these languages are object-oriented, with constraints on the object variables embedded within class definitions. These objects then map directly to components such as servers and software packages. In [11], the constraint system itself also replaces traditional type checking – instead, an object is an instance of a type if it satisfies a given set of constraints.

3 Implementations

The problem of specifying configurations, as well as distributing and applying them to their respective target systems, is already well researched, and various products exist. It might therefore be advantageous to combine constraints and constraint-solving with existing configuration systems.

For example, Hewson and Anderson [9] interface with the Puppet [17] configuration environment and language in order to leverage this already common tool.

Beside Puppet, there are several other configuration systems which offer declarative interfaces and therefore could be used in this context. Examples are CFEngine [2] (which has some more advanced features, such as automatically determining the steps required in order to update a configuration) and Chef [3].

Di Cosmo et al. also follow the idea of reutilizing existing systems in [5], where they target OpenStack. Software packages are selected from existing package repositories and tools (such as APT on Debian), depending on the target system. In the article, it is also pointed out how constraint solving (to generate configurations, select components etc.) and existing tools (to assemble components, apply configurations and deploy them) complement each other. Such a combination would ideally result in a completely automated configuration and deployment process, which could dynamically provision optimally configured resources based on a user's specific requirements.

Generally, it can be observed that all approaches utilize existing open-source constraint solvers and optimizers, such as Gecode, and often cross-solver languages (e.g. MiniZinc) to interact with them. For example, Gecode is very general constraint solving and optimization framework, and could be used for a large variety of applications. However, there is a significant drawback to these properties - these languages and systems provide rather limited expressiveness, and are not readily suited for specific domains. Gecode provides only very low-level facilities, and even commonplace expressions such as foreach constraints are not directly supported by the API.

Therefore, the languages are often extended, or used as intermediary formats generated from sources in a different language. Unfortunately, conversions from different formats might necessitate complex and computationally intensive translations, as described e.g. in [10]. Furthermore, it is necessary to define an origin format which is well-suited for the domain.

In general, it can be concluded that the challenge of declarative configuration lies not in the solving process itself (which has already been taken care of by various projects). Instead, the challenge is to provide users with accessible and intuitive ways to specify constraints, and to handle the solutions usefully (e.g. by automatic deployment).

3.1 ConfSolve

Hewson et al. [10] present ConfSolve, an "object-oriented declarative configuration language", based on constraint programming. ConfSolve has a rigorously defined type system, which supports sets, objects, and subtyping as a core concept.

ConfSolve is compiled to MiniZinc, which is further compiled to FlatZinc. This format can be parsed and solved by Gecode, which generates a flat solution (i.e. key-value pairs). The solution is parsed by the ConfSolve system to generate the final configuration in the form of an object tree. Any variable chosen by the optimizer can be set as an optimization goal (either minimization or maximization).

Additionally, validation of existing (e.g. manually edited) solutions is also supported.

In this case, it is checked whether the given configuration satisfies all applicable constraints.

Performance was found to be satisfying, with a maximum runtime of about 1.5 minutes for the largest successful instance. This involved assignments to 750 virtual machines, the maximum instance size which could be evaluated within the 4.5GB memory limit. This maximum size was found to be significantly higher than when using a comparable theorem proving/SAT approach.

3.2 Cloud Applications & Distributed Systems

Since a solution for a declarative constraint model can be found automatically, this approach is also quite interesting for automated application deployment and redeployment. Applications may be automatically configured and deployed to specific systems, and later, the configurations can be adapted to changing conditions.

In [4], Dearle, Kirby and McCarthy propose a framework for configuring autonomic distributed systems using a declarative constraint language, and finding configuration values using a constraint solver. More unusually, their system is capable of dynamically re-configuring itself (or ”self-healing”) when it detects that, e.g. due to a host going offline, the original configuration no longer fulfills the constraints.

They declaratively describe ”configuration goals”, which include required resources (both software and physical), and connections between them (e.g. latency and bandwidth). Validity of the constraints is continuously monitored, and a self-healing process is initiated if any of them are found to be violated. To heal itself, the system picks new values such that the changed constraints are satisfied again.

The flexibility gained by specifying constraints can also be harnessed at larger scales. In Di Cosmo et al.'s approach [5], the user specifies abstract services, whose concrete implementations are interchangeable and only fixed when assigned to machines.

Also, a major difference to other approaches, such as ConfSolve, is the use of ports (which are represented as constraints) for modeling dependencies at the interfaces between components. Incoming ports specify capabilities or interfaces provided by a component, e.g. to indicate that an SQL server provides a database connection. Optionally, a maximum number of components which may connect to the port can be specified (e.g. an SQL server supports at most 3 application instances). Outgoing ports declare either requirements (which indicate that at least n instances of some component are required, e.g. at least 2 database shards for an application) or conflicts (if the current component is present, some other one must not be, e.g. two components which perform the same task).

Furthermore, this implementation finds not only some solution satisfying the constraints, but attempts to find an optimal one. Two different optimization goals are supported: minimizing either the number of machines used, or the differences to an initial configuration.

Future extension possibilities proposed by the authors are the adjustment of existing configurations, life cycles/states and dynamically changing constraints or dependencies.

3.3 Service Composition

In Mayer et al. [14], constraint solving is used for configuration tasks in Service Oriented Architectures. Their approach supports dynamic service composition, thus it is not necessary to specify exact numbers of components in advance. Instead, the solver can generate them as needed.

Similar to Di Cosmo et al. [5], this system uses the ports concept. Here, however, the ports are not a special feature, but consist simply of arbitrary constraints on accepted or provided messages. These specifications are far more powerful than the simple type specifications which are traditionally used, while maintaining the clarity of ports as specifications of interdependencies.

However, one drawback compared to more specialized languages is that message directionality cannot be readily expressed and is only known by convention. Furthermore, while a core use case is the specification of workflows or message flows, the semantics are not very intuitive for this purpose.

Somewhat unusually, the authors opted to design their own solver, which uses an iterative limited-depth strategy to ensure termination.

3.4 Aspect-Based Modeling

Nechypurenko et al. [16] present an application of aspect-oriented programming to automotive distributed embedded systems. These systems typically have many complex constraints, such as severely limited resources of embedded devices, as well as physical constraints not present in other environments (e.g the distance between units). Furthermore, the systems are highly distributed, consisting of many interconnected devices, each running specialized software. Code must be specially written for each component and adapted for each deployment, obviously at high costs.

In order to reduce component costs, manufacturers increasingly utilize COTS (commercial off-the-shelf) components, but these require even more complex integration since they are more generic.

To alleviate these problems, the authors propose a move to model-driven development. Hardware configuration and software components are modeled as separate concerns, with the weaved model mapping software components to hardware units (ECUs).

The resulting constraints and interdependencies would be far too numerous and complex to manually construct a model, which would make this modelling method too complex to handle. Instead, domain-specific constraints are declared, which the final weaved model respects.

However, the users (i.e. domain experts) still struggled to use the various constraint solvers directly, through their C or Java interfaces. Instead, a domain-specific knowledge base was implemented in Prolog, which provided a more intuitive interface to the experts. Furthermore, this relieves the domain experts from having to handle technical specifics such as solving strategies.

Despite the use of a more expressive language, it was still not (easily) possible to include certain constraints, such political considerations or technical legacies. The authors therefore decided to enable fixing parts of a model, that is, manually specifying values for certain parameters.

Once all constraints have been specified, finding a valid configuration is a two-step process: first, global constraints are solved to prune the search space, then a result is calculated and optimized for some criterion (e.g. the number of ECUs used). Despite the optimization step, the solving process was found to still be quite slow. By adding domain-specific heuristics to ”guide” the solver by prioritizing assignments, performance was significantly improved.

The complexity of the constraint set can make it very hard to find errors which prevent all or only some solutions (which may not even be noticed). Even once found, correcting such errors still requires care, since potential side effects of corrections have to be considered. In order to simplify this task, developers can specify a set of modifications which the solver can apply automatically in order to repair a model. For example, additional RAM could be added if solving fails due to a component not having sufficient memory available.

3.5 Model-Driven Engineering

An approach to model-driven engineering in similar applications was proposed by White et al. [19], as an extension of their earlier work in [20], for configuring distributed real-time embedded systems such as those found in cars and airplanes. Here, the idea was to combine components to build applications only at deployment time (late binding).

The cooperation of different roles (e.g. component developers or deployers) with conflicting interests is required to integrate the final product. The compatibility of the individual component configurations is extremely important – high precision is required, since subtle mistakes can cause hard-to-diagnose failures. Again, this implies a large variety of constraints between components which must be fulfilled in order to guarantee correct functioning of the assembled system.

The authors developed a tool named ”Fresh”, which transforms feature models into constraint satisfaction problems, and derives an optimal solution according to a configurable cost function. The feature models can include constraints between features (e.g. ”ARM binaries require an ARM processor”). Such requirements may have cardinalities to indicate that more than a certain number of some component are required, or that at most a certain number may be present. It is also possible to declare a component as optional. XOR, or ”one of”, constraints, are supported as well. It is possible to control multiple configuration steps and activities, application packaging, etc. The feature models are reduced to CSPs, then solved, and the resulting configuration settings (e.g. component assignments and the matching machine architectures for compiling) are automatically exported to XML.

An avionics application based on CORBA components is presented as a working example. Using the presented system, the size of configurations could be significantly reduced ($> 90\%$ in the given example), and reconfiguration was dramatically simplified. However, the example mentioned in the article is relatively small, and no information on performance or run-time is given.

3.6 Related Work

Among other works related to this field are Hinrichs et al.'s approach [11], in which a first-order theorem prover is used to solve object-oriented CSPs (which are converted to first-order logical clauses). Theorem provers have the advantage of being very general, as well as highly optimized, but on the other hand, transformation of constraints into clausal form is required.

Fischer et al.'s Engage system [6] for deployment management also generates configurations using a constraint solver. In addition, it includes a run-time part to deploy configurations, start configured services, and perform upgrades.

Liu et al.'s COPE [13] is intended to automate cloud orchestration using declarative policies. It takes a set of policies and a description of the current system state as inputs, then attempts to generate a new, optimized state in accordance with the policy.

As mentioned above, the correctness of the generated configurations hinges on the correctness of the constraints. To improve the latter, Nadi et al. [15] propose to extract constraints directly from software. A different approach followed by Kiciman and Wang [12] is the extraction of constraints from existing, tested configurations.

4 Conclusion

Overall, the use of constraint programming in configuration applications appears as a promising concept, which is especially relevant in today's increasingly complex cloud infrastructures.

However, adoption for use in production seems to be rather slow. Of the papers surveyed, few cited concrete industrial applications (e.g. [5]). Furthermore, there seems to be surprisingly little activity in the ”core” software domains, with many applications of constraint solving to configuration coming from the embedded and hardware sectors (e.g. automotive).

While several declarative configuration systems (such as Puppet, CFEngine or Chef) exist, they still largely require manual configuration – there is no product with comparable impact implementing the concepts presented here.

As seen in the approaches described, many existing components (such as constraint languages and solvers, configuration tools, package management systems, deployment mechanisms) can be readily used, which significantly accelerates development and may significantly reduce migration costs.

However, many of the limitations which were discovered in the development and use of expert systems more than 20 years ago still persist – most notably, the difficulty of capturing complex domain constraints and translating them to declarative models, especially when done by domain (not software) experts (see e.g. [16]).

Common problems of constraint programming, such as hard-to-diagnose errors [16], and, for optimization, having to define a cost function considering various trade-offs, apply as well.

Overall, however, the idea of automatically generating configurations from constraints offers several unique opportunities, such as significant reductions in complexity and workload, and automatic optimization of configurations. Additionally, it may be seen as a logical continuation of existing declarative configuration languages.

References

[1] Virginia E. Barker, Dennis E. O'Connor, Judith Bachant, and Elliot Soloway. Expert systems for configuration at digital: XCON and beyond. *Commun. ACM*, 32(3):298–318, March 1989.

[2] Mark Burgess et al. Cfengine: a site configuration engine. In *USENIX Computing systems*, 1995.

[3] Chef Software. Chef. `https://www.getchef.com/chef/`. Accessed: 2014-11-18.

[4] Alan Dearle, Graham NC Kirby, and Andrew J McCarthy. A framework for constraint-based development and autonomic management of distributed applica-

tions. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 300–301. IEEE, 2004.

[5] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 211–222. ACM, 2014.

[6] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. *ACM SIGPLAN Notices*, 47(6):263–274, 2012.

[7] Gerhard Fleischanderl, Gerhard E Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large systems using generative constraint satisfaction. *IEEE Intelligent Systems*, 13(4):59–68, 1998.

[8] Gecode Team. Gecode: Generic constraint development environment. `http://www.gecode.org/`. Accessed: 2014-12-21.

[9] John A Hewson and Paul Anderson. Modelling system administration problems with CSPs. *Constraint Modelling and Reformulation (ModRef'11)*, 2011.

[10] John A Hewson, Paul Anderson, and Andrew D Gordon. A declarative approach to automated configuration. In *LISA*, volume 12, pages 51–66, 2012.

[11] Tim Hinrichs, Nathaniel Love, Charles Petrie, Lyle Ramshaw, Akhil Sahai, and Sharad Singhal. Using object-oriented constraint satisfaction for automated configuration generation. In *Utility Computing*, pages 159–170. Springer, 2004.

[12] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 28–35. IEEE, 2004.

[13] Changbin Liu, Boon Thau Loo, and Yun Mao. Declarative automated cloud resource orchestration. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 26. ACM, 2011.

[14] Wolfgang Mayer, Rajesh Thiagarajan, and Markus Stumptner. Service composition as generative constraint satisfaction. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 888–895. IEEE, 2009.

[15] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Mining configuration constraints: static analyses and empirical results. In *ICSE*, pages 140–151, 2014.

[16] Andrey Nechypurenko, Egon Wuchner, Jules White, and Douglas C Schmidt. Application of aspect-based modeling and weaving for complexity reduction in the development of automotive distributed realtime embedded system. In *Proceedings of the Sixth International Conference on Aspect-Oriented Software Development*, 2007.

[17] Puppet Labs. Puppet. `http://puppetlabs.com/puppet`. Accessed: 2014-11-18.

[18] Markus Stumptner, Alois Haselböck, and Gerhard Friedrich. COCOS - a tool for constraint-based, dynamic configuration. In *Artificial Intelligence for Applications, 1994., Proceedings of the Tenth Conference on*, pages 373–380. IEEE, 1994.

[19] Jules White and Douglas C Schmidt. Automated configuration of component-based distributed real-time and embedded systems from feature models. In *17th Annual Conference of the International Federation of Automatic Control, Seoul, Korea*, 2008.

[20] Jules White, Douglas C Schmidt, Krzysztof Czarnecki, Christoph Wienands, and Gunther Lenz. Automated model-based configuration of enterprise java applications. In *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*, pages 301–301. IEEE, 2007.