Solving Combined Configuration Problems: A Heuristic Approach[1]

Martin Gebser[2] and **Anna Ryabokon**[3] and **Gottfried Schenner**[4]

Abstract. This paper describes an abstract problem derived from a combination of Siemens product configuration problems encountered in practice. Often isolated parts of configuration problems can be solved by mapping them to well-studied problems for which efficient heuristics exist (graph coloring, bin-packing, etc.). Unfortunately, these heuristics may fail to work when applied to a problem that combines two or more subproblems. In the paper we show how to formulate a combined configuration problem in Answer Set Programming (ASP) and to solve it using heuristics à la hclasp. The latter stands for heuristic clasp that is nowadays integrated in clasp and enables the declaration of domain-specific heuristics in ASP. In addition, we present a novel method for heuristic generation based on a combination of greedy search with ASP that allows to improve the performance of clasp.

1 Introduction

Researchers in academia and industry have tried different approaches to configuration knowledge representation and reasoning, including production rules, constraints languages, heuristic search, description logics, etc.; see [16, 14, 9] for surveys. Although constraint-based methods remain de facto standard, Answer Set Programming (ASP) has gained much attention over the last years because of its expressive high-level representation abilities.

As evaluation shows ASP is a compact and expressive method to capture configuration problems [15, 18, 9], i.e. it can represent configuration knowledge consisting of component types, associations, attributes, and additional constraints. The declarative semantics of ASP programs allows a knowledge engineer to choose the order in which rules are written in a program, i.e. the knowledge about types, attributes, etc. can be easily grouped in one place and modularized. Sound and complete solving algorithms allow to check a configuration model and support evolution tasks such as reconfiguration. Generally, the results prove that ASP has limitations when applied to large-scale product (re)configuration instances [1, 5]. The best results in terms of runtime and solution quality were achieved when domain-specific heuristics were applied [17, 12].

In this paper we introduce a combined configuration problem that reflects typical requirements frequently occurring in practice at Siemens. The parts of this problem correspond (to some extent) to classical computer science problems for which there already exist some well-known heuristics and algorithms that can be applied to speed up computations and/or improve the quality of solutions.

As the main contribution, we present a novel approach on how heuristics generated by a greedy solver can be incorporated in an ASP program to improve computation time (and obtain better solutions). The application of domain-specific knowledge formulated succinctly in an ASP heuristic language [8] allows for better solutions within a shorter solving time, but it strongly deteriorates the search process when some additional requirements (conflicting with the formulated heuristics) are included. On the other hand, the formulation of complex heuristics might be cumbersome using greedy methods. Therefore, we exploit a combination of greedy methods with ASP for the generation of heuristics and integrate them to accelerate an ASP solver. We evaluate the method on a set of instances derived from configuration scenarios encountered by us in practice and in general. Our evaluation shows that for three different sets of instances solutions can be computed an order of magnitude faster than compared to a plain ASP encoding.

The remainder of this paper is structured as follows. Section 2 introduces a combined configuration problem (CCP) which is exemplified in Section 3. Section 4 discusses heuristics for solving the CCP. We present our evaluation results in Section 5. Finally, in Section 6 we conclude and discuss future work.

2 Combined Configuration Problem

The Combined Configuration Problem (CCP) is an abstract problem derived from a combination of several problems encountered in Siemens practice (railway interlocking systems, automation systems, etc.). A CCP instance is defined by a directed acyclic graph, called just graph later on in this paper for simplicity. Each vertex of the graph has a type and each type of the vertices has a particular size. In addition, each instance comprises two sets of vertices specifying two vertex-disjoint paths in the graph. Furthermore, an instance contains a set of areas, sets of vertices defining possible border elements of each area and a maximal number of border elements per area. Finally, a number of available colors as well as a number of available bins and their capacity are given.

Given a CCP instance, the goal is to find a solution that satisfies a set of requirements. All system requirements are separated into the corresponding subproblems which must be solved together or in combinations:

- **P1 Coloring** *Every vertex must have exactly one color.*
- **P2 Bin-Packing** *For every color a Bin-Packing problem must be solved. For every color the same number of bins are available. Every vertex must be assigned to exactly one bin of its color and*

[1] This work was funded by COIN and AoF under grant 251170 as well as by FFG under grant 840242. An extended version of this paper is to appear in the proceedings of the 13th International Conference on Logic Programming and Non-monotonic Reasoning.
[2] Aalto University, HIIT, Finland and University of Potsdam, Germany, email: martin.gebser@aalto.fi
[3] Alpen-Adria-Universität Klagenfurt, Austria, email: anna.ryabokon@aau.at
[4] Siemens AG Österreich, Austria, email: gottfried.schenner@siemens.com

Juha Tiihonen, Andreas Falkner and Tomas Axling, Editors
Proceedings of the 17th International Configuration Workshop
September 10-11, 2015, Vienna, Austria

for every bin it holds that the sum of sizes must be smaller or equal to the bin capacity.

- **P3 Disjoint Paths** *Vertices of different paths cannot be colored in the same color.*
- **P4 Matching** *Each border element must be assigned to exactly one area such that the number of selected border elements of an area does not exceed the maximal number of border elements and all selected border elements of an area have the same color.*
- **P5 Connectedness** *Two vertices with the same color must be connected via a path that contains only vertices of that color.*

Origin In the railway domain the given graph represents a track layout of a railway line. A coloring **P1** can then be thought as an assignment of resources (e.g. computers) to the elements of the railway line. In real-world scenarios different infrastructure elements may require different amounts of a resource that is summarized in **P2**. This may be hardware requirements (e.g. a signal requiring a certain number of hardware parts) or software requirements (e.g. an infrastructural element requiring a specific processing time). The requirements of **P1** and **P2** are frequently used in configuration problems during an assignment of entities of one type to entities of another type [11, 5].

The constraint of **P3** increases availability, i.e. in case one resource fails it should still be possible to get from a source vertex (no incoming edges) of the graph to a target vertex (no outgoing edges) of the graph. In the general version of this problem one has to find n paths that maximize availability. The CCP uses the simplified problem where 2 vertex-disjoint paths are given.

P4 stems from detecting which elements of the graph are occupied. The border elements function as detectors for an object leaving or entering an area. The Partner Units Problem [2, 1] is a more elaborate version of this problem. **P5** arises in different scenarios, e.g. if communication between elements controlled by different resources is more costly, then neighboring elements should be assigned to the same resource whenever possible.

3 Example

Figure 1 shows a sample input CCP graph. In this section we illustrate how particular requirements can influence a solution. Namely, we add the constraints of each subproblem one by one. If only **P1** is active, any graph corresponds to a trivial solution of **P1** where all vertices are colored white.

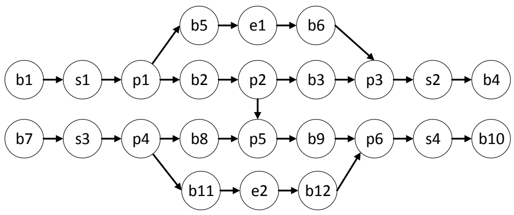


Figure 1: Input CCP graph and a trivial solution of Coloring (**P1**)

Let us consider the input graph as a Bin-Packing problem instance with four colors and three bins per color of a capacity equal to five. The vertices of type b, e, s and p have the sizes 1, 2, 3 and 4, respectively. A solution of Coloring and Bin-Packing (**P1-P2**) is presented in Figures 2 and 3.

Moreover, two vertex-disjoint paths are declared by $path1 = \{b1, s1, p1, b2, p2, b3, p3, s2, b4\}$ as well as $path2 = \{b7, s3, p4, b8, p5, b9, p6, s4, b10\}$, and the Disjoint Paths constraint (**P3**) is active. Consequently, in this case the solution shown in Figure 2 violates this constraint and must be modified as given in Figure 4 where the vertices of different paths are colored with different colors ($path1$ with dark grey and grey, and $path2$ with white and light grey).

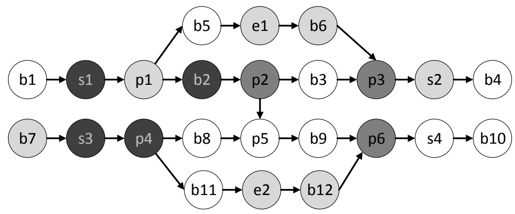


Figure 2: Used colors in a solution of the Coloring and Bin-Packing problems (**P1**-**P2**)

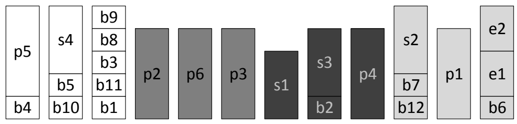


Figure 3: Used bins in a solution of the Coloring and Bin-Packing problems (**P1**-**P2**)

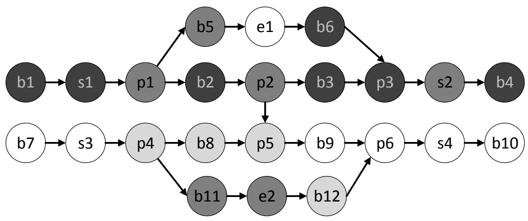


Figure 4: Solution of the Coloring, Bin-Packing and Disjoint Paths problems (**P1**-**P3**)

Figure 5 shows an example of Matching (**P4**). In this example there are seven areas in the matching input graph, each corresponding to a subgraph surrounded with border elements (Figure 1). For instance, area $a1$ represents the subgraph $\{b1, s1, p1, b2, b5\}$ and area $a2$ the subgraph $\{b5, e1, b6\}$. The corresponding border elements, $\{b1, b2, b5\}$ and $\{b5, b6\}$, are displayed in Figure 5.

Assume that an area can have at most 2 border elements assigned to it. In the resulting matching (Figure 5) $b1$, $b2$ are assigned to $a1$, whereas $b5$, $b6$ are assigned to $a2$. Note that the sample selected matching shown in Figure 5 is not valid with the coloring presented previously, because, for example, $b5$ and $b6$ are assigned to the same area $a2$ although they are colored differently.

In addition, the coloring solution shown in Figure 4 violates the Connectedness constraint (**P5**). Therefore, the previous solutions must be updated to take the new requirements into account. Figure 6 shows a valid coloring of the given graph that satisfies all problem requirements (**P1**-**P5**).

4 Combining Heuristics for Configuration Problems

We formulated the CCP using ASP and the corresponding encoding can be found at `http://isbi.aau.at/hint/problems`.

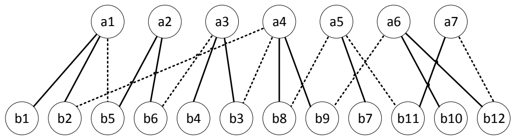


Figure 5: A sample input and solution graphs for **P4**. The selected edges of the input graph are highlighted with solid lines.

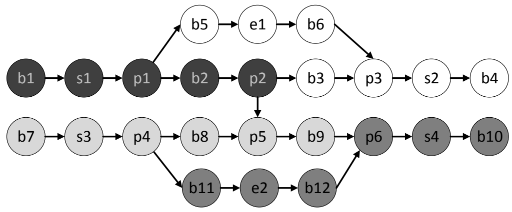


Figure 6: A valid solution for **P1**-**P5**

To formulate a heuristic within ASP we use the declarative heuristic framework developed by Gebser et al. [8]. In this formalism the heuristics are expressed using atoms $\_heuristic(a, m, v, p)$, where a denotes an atom for which a heuristic value is defined, m is one of four modifiers (init, factor, level and sign), and v, p are integers denoting a value and a priority, respectively, of the definition. A number of shortcuts are available, e.g. $\_heuristic(a, v, l)$, where a is an atom, v is its truth value and l is a level. The heuristic atoms modify the behavior of the VSIDS heuristic [10]. Thus, if a $\_heuristic$ atom is true in some interpretation, then the corresponding atom a might be preferred by the ASP solver at the next decision point.

There are different ways to incorporate heuristic atoms in a program. The standard approach [8] requires an implementation of a heuristic at hand using a pure ASP encoding, whereas the idea of our method is to delegate the (expensive) generation of a heuristic to an external tool and then to extend the program with generated heuristic atoms to accelerate the ASP search. Below we will exemplify how both approaches can be applied.

4.1 Standard generation of heuristics in ASP

Several heuristics can be used for the problems that compose the CCP. For instance, for the coloring of vertices (**P1**) we seek to use as few colors as possible by the following rule:

```
_heuristic(vertex_color(V,C),true,MC-C) :-
    vertex(V), color(C), nrofcolors(MC).
```

Listing 1: Heuristic for an assignment of colors to vertices

Roughly speaking, this rule means that the assignment of colors to vertices must be done in an ascending order of colors. Given a $vertex(b1)$ and two colors $nrofcolors(2)$ encoded as $color(1)$ and $color(2)$, the solver can derive two heuristic atoms $\_heuristic(vertex\_color(b1, 1), true, 1)$ and $\_heuristic(vertex\_color(b1, 2), true, 0)$. These atoms indicate the solver that the atom $vertex\_color(b1, 1)$ must be assigned the truth value $true$ first since the atom with the higher level is preferred.

Additionally, we can apply the well-known Bin-Packing heuristics for the placement of colored vertices into the bins of specified capacity (**P2**). The Bin-Packing problem is known to be an NP-hard combinatorial problem. However, there are a number of approximation algorithms (construction heuristics) that allow efficient computation of good approximations of a solution [6], e.g. Best/First/Next-Fit heuristics. They can, of course, be used as heuristics for the CCP.

Let the Bin-Packing problem instance be encoded using a set of predicates among which $nrofbins/1$ and $order/2$ denote a number of bins in the instance and an ordered set of input vertices, respectively. The predicate $vertex\_bin/2$ is used to encode a solution and denotes an assignment of a vertex to a bin. As shown in Listing 2, given a (decreasing) order of vertices, we can force the solver to place vertex V_i into the lowest-indexed bin for which the size of already placed vertices does not exceed the capacity, i.e. in a first-fit bin. The heuristic never uses a new bin until all the non-empty bins are full and it can be expressed by rules that generate always a higher level for the bins with smaller number:

```
binDomain(1..NB) :- nrofbins(NB).
offset(NB+1) :- nrofbins(NB).
_heuristic(vertex_bin(V,B),true,M+O*NB-B) :-
    binDomain(B), nrofbins(NB), order(V,O),
    offset(M).
```

Listing 2: First-Fit heuristic for an assignment of vertices to bins

It is also possible (with an intense effort) to express other heuristics for **P1**-**P5** that guide the search appropriately and allow to speed up the computation of solutions if we solve these problems separately. However, as our experiments show, the inclusion of heuristics for different problems at the same time might drastically deteriorate the performance for real-world CCP instances.

4.2 Greedy Search

From our observations in the context of product configuration, it is relatively easy to devise a greedy algorithm to solve a part of a configuration problem. This is often the case in practice, because products are typically designed to be easily configurable. The hard configuration instances usually occur when new constraints arise due to the combination of existing products and technologies.

The same can be said for the CCP. Whereas it is easy to develop greedy search algorithms for the individual subproblems, it becomes increasingly difficult to come up with an algorithm that solves the combined problem. For instance, a greedy algorithm for the Matching problem of the CCP (**P4**) can be formulated as follows: For every vertex v find a related area a with the fewest assigned vertices so far and match v with a. The algorithm assumes that all border elements are colored with one color, as it trivially satisfies the coloring requirement of the matching problem. A greedy algorithm for solving the CCP wrt. Coloring, Bin-Packing and Connectedness (**P1**, **P2** and **P5**) can be described as follows:

1. Select the first available color c and add the first vertex not assigned to any bin to a queue Q;
2. Get and remove from Q the first element v, label it with c and try to assign it to a bin using some Bin-Packing heuristic, e.g. First-Fit or Best-Fit [6];
3. If v is assigned to some bin, add neighbors of v to Q;
4. If $Q \neq \emptyset$, then goto 2;
5. Otherwise, if there are unassigned vertices, then make the color c unavailable and goto 1.

Suppose one wants to combine these two algorithms. One strategy would be to run greedy Matching and then solve the Bin-Packing

Algorithm 1: Greedy & ASP

Input: A problem P, an ASP program Π solving the problem P
Output: A solution S

$GreedySolution \leftarrow$ `solveGreedy`(P);
$H \leftarrow$ `generateHeuristic`$(GreedySolution)$;
return `solveWithASP`(Π, H);

problem taking matchings into account. Namely, the combined algorithm preforms the following steps:

1. Call the matching greedy algorithm and get a set of matchings $M = \{(v_1, a_1), \ldots, (v_n, a_m)\}$;
2. For each vertex v_i of the input graph G do:
   (a) Assign a new color to v_i, if v_i has no assigned color;
   (b) Put v_i into a bin, as in the greedy Bin-Packing (steps 2-3);
   (c) If v_i is a border element, then retrieve an area a_j that matches v_i in M and color all vertices of this area in the same color as v_i.

The combined algorithm might violate Connectedness, because it colors all border vertices assigned to an area with the same color. However, these vertices are not necessarily connected. That is, there might be a solution with a different matching, but the greedy algorithm tests only one of all possible matchings. Moreover, there is no obvious way how to create an algorithm solving all three problems efficiently. This is a clear disadvantage of using ad-hoc algorithms in contrast to the usage of a logic-based formalism like ASP, where the addition of constraints is just a matter of adding some rules to an encoding. On the other hand, domain-specific algorithms are typically faster and scale better than ASP-based or SAT-based approaches that cannot be used for large instances. For instance, the memory demand of the greedy Bin-Packing algorithm is polynomial in graph size.

4.3 Combining Greedy Search and ASP

One way to let a complete ASP solver and a greedy search algorithm benefit from each other is to use the greedy algorithm to compute upper bounds for the problem to solve. The tighter upper bound usually means smaller grounding size and shorter solving time, because the greedy solver being domain-specific usually outperforms ASP for the relaxed version of the problem. For instance, running the greedy algorithm for the Bin-Packing problem and Matching problem gives upper bounds for the maximal number of colors, i.e. number of different Bin-Packing problems to solve. The same applies to the Matching problem. This kind of application of greedy algorithms has a long tradition in branch and bound search algorithms, where greedy algorithms are used to compute the upper bound of a problem. For an example see [19], where a greedy coloring algorithm is used to find an upper bound for the clique size in a graph for the computation of maximum cliques. In this paper we investigate a novel way to combine greedy algorithms and ASP (Algortihm 1). Consequently, in our approach we, first, use a greedy algorithm to find a solution of a relaxed version of the problem. Next, this solution is converted into a heuristic for an ASP solver which assigns the atoms of the greedy solution a higher heuristic value.

As an example for solving the complete CCP problem, we can, first, find an unconnected solution for the combination of Coloring, Bin-Packing, Disjoint paths and Matching problems (**P1**-**P4**), and then, use the ASP solver to fix the Connectedness property (**P5**). The idea of combining local search with a complete solver is also found in large neighborhood search [4].

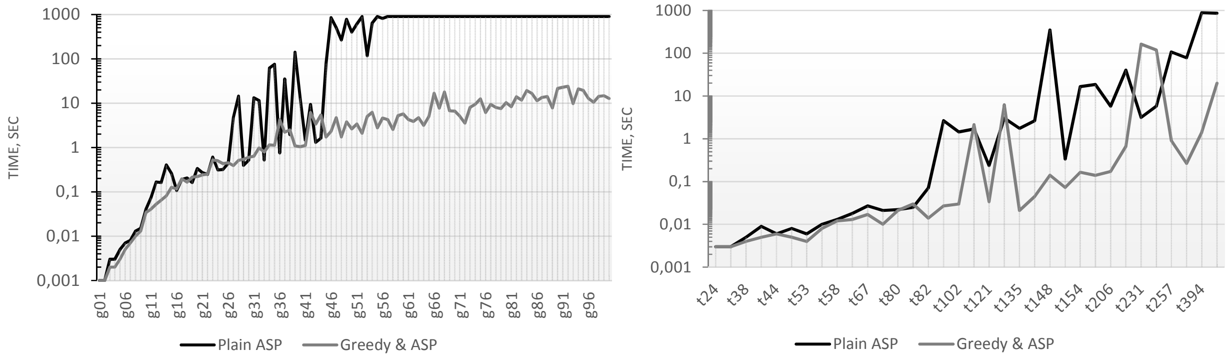
5 Experimental results

Experiment1 In our evaluation we compared a plain ASP encoding of the CCP with an ASP encoding extended with domain-specific knowledge. The Bin-Packing problem (**P2**) of the CCP corresponds to the classic Bin-Packing problem and the same heuristics can be applied. We implemented several Bin-Packing heuristics such as First/Best/Next-Fit (Decreasing) heuristics using ASP as shown in Section 4.1. For the evaluation we took 37 publicly available Bin-Packing problem instances[5], for which the optimal number of bins $optnrofbins$ is known, and translated them to CCP instances. The biggest instance of the set includes 500 vertices and 736 bins of the capacity 100. In the experiment, the maximal number of colors was set to 1 and the maximal number of bins was set to $2 \cdot optnrofbins$. All instances were solved by both approaches[6]. For a plain ASP encoding the solver required at most 27 seconds to find a solution whereas for the heuristic ASP program solving took at most 6 seconds, which is 4.5 times faster. The best results for the heuristic approach were obtained using the First-Fit heuristic with the decreasing order of vertices. Corresponding solutions utilized less bins then the ones obtained with the plain ASP program. Moreover, using First-Fit heuristic, for 23 from 37 instances a solution with optimal number of bins was found and for 13 other instances at most 4 bins more were required. The plain ASP encoding resulted in solutions that used on average 4 bins more than corresponding solutions of the heuristic approach.

Experiment2 In the next experiment we tested the same Bin-Packing heuristics implemented in ASP for the combined CCP, i.e. when all subproblems **P1**-**P5** are active, on 100 real-world test instances of moderate size (maximally 500 vertices in an input). The instances in this experiment were derived from a number of industrial configuration tasks. Neither the plain program nor the heuristic program were able to improve runtime/quality of solutions. Moreover, our greedy method described in Section 4.2 also failed to find a connected solution, i.e. when **P5** is active. For this reason, we investigated the combined approach (Greedy & ASP) described in Section 4.3. This approach uses the greedy method to generate a partial solution ignoring the Connectedness constraint and provides this solution as $\_heuristic$ atoms to the ASP solver. Our experiments show (see Figure 7a) that the combined approach can solve all 100 benchmarks from the mentioned set, whereas the plain encoding solves only 54 instances (the time frame was set to 900 seconds in this and the next experiment). Moreover, for those instances which were solved using both approaches, the quality of solutions measured in terms of used bins and colors was the same. However, the runtime of the combined approach was 18 times faster on average and required at most 24 seconds instead of 848 seconds needed for the plain ASP encoding.

Experiment3 In addition, we tested more complex real-world instances (maximally 1004 vertices in an input)[7] which we have also submitted to the ASP competition 2015. Similarly to *Experiment2*

[5] http://www.wiwi.uni-jena.de/Entscheidung/binpp/index.htm
[6] The evaluation was performed using clingo version 4.3.0 from the Potassco ASP collection [7] on a system with Intel i7-3030K CPU (3.20 GHz) and 64 GB of RAM, running Ubuntu 11.10.
[7] The instances are available at: http://isbi.aau.at/hint/problems



(a) *Experiment2* (b) *Experiment3*

Figure 7: Evaluation results using Plain ASP and Greedy & ASP

we compared the plain ASP encoding to the combined approach from Section 4.3. Again, regarding the quality of solutions, both approaches are comparable, i.e. they use on average the same number of colors and bins, with the combined approach having a slight edge. Generally, from 48 instances considered in this experiment, 36/38 instances were solved using the plain/combined encoding, respectively. On average/maximally the plain encoding needed 69/887 seconds to find a solution whereas the combined method took 14/196 seconds, respectively, which is about 5 times faster. Figure 7b shows the influence of heuristics on the performance for the instances from *Experiment3* that were solved by both approaches within 900 seconds. Although the grounding time is not presented for both experiments, we note that it requires about 10 seconds using both approaches for the biggest instance when all subproblems **P1**-**P5** are active.

6 Discussion

Choosing the right domain-specific heuristics for simple backtrack-based solvers is essential for finding a solution at all, especially for large and/or complex problems. The role of domain-specific heuristics in a conflict-driven nogood learning ASP solver seems to be less important when it comes to solving time. Here the size of the grounding and finding the right encoding is often the limiting factor. Nevertheless, *domain-specific heuristics are very important* to control the order in which answer sets are found and are an alternative to optimization statements. As we have shown, domain-specific heuristics also provide a mechanism to combine greedy algorithms with ASP solvers, which opens up the possibility to use ASP in a meta-heuristic setting. However, the possible applications go beyond this. The same approach could be used to repair an infeasible assignment using an ASP solver. This is currently a field of active research for us and has applications in the context of product reconfiguration. Reconfiguration occurs when a configuration problem is not solved from scratch, but some parts of an existing configuration have to be taken into account.

An open question is how to combine heuristics for different subproblems in a modular manner without the adaptation of every domain-specific heuristic. Here approaches like search combinators [13] from the constraint programming community might be useful. Another interesting topic for future research would be how to learn heuristics from an ASP solver, i.e. to investigate the variable/value order chosen by an ASP solver for medium size problem instances and use heuristics in a backtrack solver for larger instances that are out of scope of an ASP solver due to the grounding size. Some aspects of this topic were discussed in [3].

REFERENCES

[1] M. Aschinger, C. Drescher, G. Friedrich, G. Gottlob, P. Jeavons, A. Ryabokon, and E. Thorstensen, 'Optimization Methods for the Partner Units Problem', in *Proceedings of CPAIOR*, pp. 4–19, (2011).

[2] M. Aschinger, C. Drescher, G. Gottlob, P. Jeavons, and E. Thorstensen, 'Tackling the Partner Units Configuration Problem', in *Proceedings of IJCAI*, pp. 497–503, (2011).

[3] M. Balduccini, 'Learning and using domain-specific heuristics in ASP solvers', *AI Communications*, **24**(2), 147–164, (2011).

[4] Raffaele Cipriano, Luca Di Gaspero, and Agostino Dovier, 'A hybrid solver for large neighborhood search: Mixing gecode and easylocal++', in *Hybrid metaheuristics*, 141–155, Springer, (2009).

[5] G. Friedrich, A. Ryabokon, A. A. Falkner, A. Haselböck, G. Schenner, and H. Schreiner, '(Re) configuration based on model generation', in *Proceedings of LoCoCo*, pp. 26–35, (2011).

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

[7] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, Morgan & Claypool Publishers, 2012.

[8] M. Gebser, B. Kaufmann, J. Romero, R. Otero, T. Schaub, and P. Wanko, 'Domain-Specific Heuristics in Answer Set Programming', in *Proceedings of AAAI*, (2013).

[9] L. Hotz, A. Felfernig, M. Stumptner, A. Ryabokon, C. Bagley, and K. Wolter, 'Configuration Knowledge Representation and Reasoning', *Knowledge-Based Configuration: From Research to Business Cases*, 41–72, (2014).

[10] C.F. Madigan, S. Malik, M.W. Moskewicz, L. Zhang, and Y. Zhao, 'Chaff: Engineering an efficient SAT solver', in *Proceedings of DAC*, (2001).

[11] W. Mayer, M. Bettex, M. Stumptner, and A. Falkner, 'On solving complex rack configuration problems using CSP methods', in *Proceedings of the IJCAI Workshop on Configuration*, (2009).

[12] A. Ryabokon, G. Friedrich, and A. A. Falkner, 'Conflict-Based Program Rewriting for Solving Configuration Problems', in *Proceedings of LPNMR*, pp. 465–478, (2013).

[13] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P. J. Stuckey, 'Search combinators', *Constraints*, **18**(2), 269–305, (2013).

[14] C. Sinz and A. Haag, 'Configuration', *IEEE Intelligent Systems*, **22**(1), 78–90, (2007).

[15] T. Soininen, I. Niemelä, J. Tiihonen, and R. Sulonen, 'Representing configuration knowledge with weight constraint rules', in *Proceedings of the Workshop on ASP*, pp. 195–201, (2001).

[16] M. Stumptner, 'An overview of knowledge-based configuration', *AI Communications*, **10**(2), 111–125, (1997).

[17] E. C. Teppan, G. Friedrich, and A. A. Falkner, 'QuickPup: A Heuristic Backtracking Algorithm for the Partner Units Configuration Problem', in *Proceedings of IAAI*, pp. 2329–2334, (2012).

[18] J. Tiihonen, M. Heiskala, A. Anderson, and T. Soininen, 'WeCoTin - A practical logic-based sales configurator', *AI Communications*, **26**(1), 99–131, (2013).

[19] Etsuji Tomita and Toshikatsu Kameda, 'An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments', *Journal of Global Optimization*, **37**(1), 95–111, (2007).