A Heuristic, Replay-based Approach for Reconfiguration

Alois Haselböck and **Gottfried Schenner** [1]

Abstract. Reconfiguration is an important aspect of industrial product configuration. Once an industrial artefact has been built according to an initial configuration, constant reconfigurations are necessary during its lifetime due to changed requirements or a changed product specification. This reconfigurations should affect as few parts of the running system as possible. Due to the large number of involved components, approaches based on optimization are often not usable in practice. This paper introduces a novel approach for reconfiguration based on a replay heuristic (the product is rebuilt from scratch while trying to use as many decisions from the legacy configuration as possible) and describes its realisation using the standard solving technologies Constraint Satisfaction and Answer Set Programming.

1 INTRODUCTION

Configuration is the task of deriving a valid, complete and purposeful system structure assembled from a set of components [14]. For non-trivial configuration problems, like product configuration, we distinguish the following levels of models (cf. Table 1): the language used to represent and solve the configuration problem (M4), the problem domain model (M3), the problem instance model (M2), and the configuration model (M1).

Table 1. Different levels of models in a configuration application.

M4	Modelling Language	e.g. UML class diagrams and OCL, CSP, ASP, ...
M3	Problem Domain Model	Generic specification of the component catalogue
M2	Problem Instance Model	Requirements specification of a concrete configuration problem
M1	Configuration Model	Solution to M2: a configuration object network

M3 is a generic specification of the problem domain. In an object-oriented environment, this would be the class model. Constraints are usually used to describe the different dependencies and restrictions between the different objects. Such a model M3 defines the space of all the technically feasible configuration solutions. Model M2 is a concrete problem instance, containing specific requirement definitions which are usually formalized in terms of constraints, initial configuration objects and requirement and resource parameters. M2 is based on M3 and uses its language and concepts (M4). Finally, M1 - a configuration - consists of all the instances, their properties and relationships realizing a final system. If this configuration is complete and consistent w.r.t. M3 and M2, M1 is said to be a solution of the given configuration problem.

[1] Siemens AG Österreich, Corporated Technology, Vienna, Austria, {alois.haselboeck, gottfried.schenner}@siemens.com

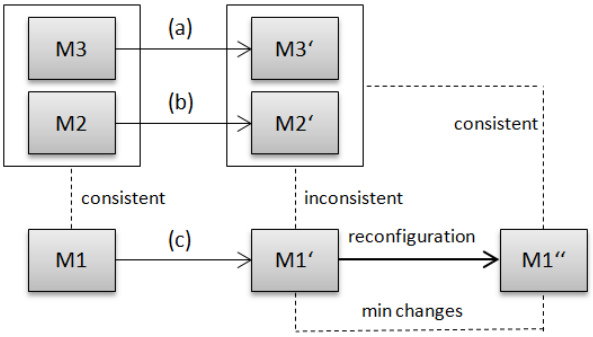


Figure 1. Reconfiguration scenarios.

Reconfiguration is the task of changing a given configuration. Various scenarios are possible why a (partial) configuration becomes inconsistent and reconfiguration is necessary (cf. Figure 1):

(a) The problem domain M3 has been changed. Reasons could be changes in the product catalogue, changes in the structure of the product line, regulation changes, etc. A legacy configuration already installed in the field may be inconsistent now to the new problem domain description and must be reconfigured.
(b) The requirements in M2 has been changed or extended. Again, a legacy configuration which is inconsistent now w.r.t. the changed requirements must be adapted.
(c) A configuration (M1) has been changed by an external process (e.g. by a manual user input or by reading a configuration from an external repository) and is now inconsistent. Again, reconfiguration must find a consistent modification of the configuration.

In all these cases, a crucial demand is that the reconfigured solution is as close as possible to the original configuration. The definition of the quality of a reconfiguration (How close is the new configuration to the legacy configuration?) could get quite subtle. [Friedrich et al., 2011], e.g., use cost functions for the different change operators. Reconfiguration is then the problem of minimizing the overall modification costs. In this paper, we are using a more light-weight approach: We don’t define cost functions but use a rather heuristic and simple definition of minimality: the number of differences between the original and the reconfigured solution should be as small as possible. This corresponds to equal cost values for all types of modifications.

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

We present methods how such reconfiguration problems can be modelled and solved by variations of standard, complete solving techniques (like SAT solving or backtracking). A challenge here is that reconfiguration starts with an inconsistent configuration fragment and standard solving (e.g. backtracking) would immediately return a `failure` result. Our idea is to start solving from scratch, but trying to re-build the search tree following the decisions of a given (inconsistent) legacy configuration. That's why we call it *replay-based* reconfiguration. The composition of a reconfiguration will deviate from the legacy configuration in cases where inconsistencies are to be avoided.

Our main contributions in this work are: (1) An Answer Set Programming (ASP) encoding of the reconfiguration problem using the special predicate `_heuristic` of `clingo` (Potassco [12]). (2) A CSP encoding of the reconfiguration problem using a novel value ordering heuristic which prefers value assignments from a legacy configuration. (3) Experimental evaluation and indications of up to which problem sizes these methods are applicable.

The rest of the paper is organized as follows: Section 2 sketches a small hardware configuration problem which will serve as example for the subsequent sections. Section 3 describes how the task of reconfiguration can be modelled and solved in 2 different frameworks: ASP and standard CSP. We compare and evaluated these techniques in Section 4 and conclude the work with a discussion of related works and a conclusion.

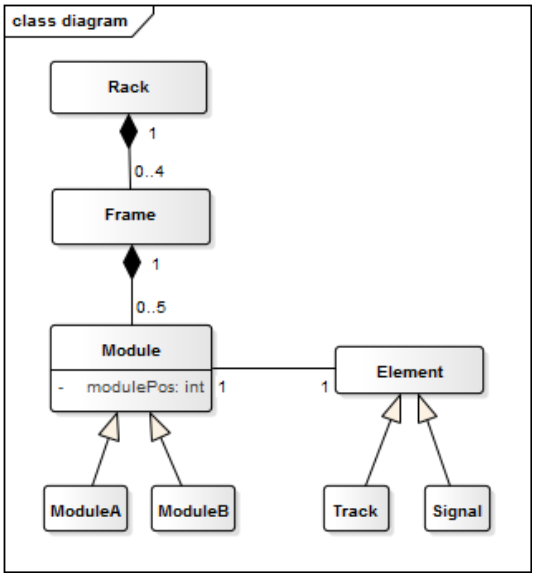


Figure 2. Problem domain model (M3) of a hardware configuration problem example.

2 EXAMPLE

A small example from the domain of railway interlocking hardware should demonstrate the dynamics of the configuration problems we want to model and solve. Of course, real-world problems are much larger and the object network and dependencies between the objects are more complex and varied.

Figure 2 shows the UML diagram and represents the problem domain M3 (cf. Table 1). A part of configuring an interlocking system is to create appropriate control modules for each outdoor element (e.g., a signal or a switch point) and to place them into the right slots of a frame, which in turn must be inserted into a rack. At the beginning, only the outdoor elements are known. Racks, frames and modules must be created during solving. In our example tracks require modules of type 'ModuleA' and signals require modules of type 'ModuleB'.

A concrete problem instance is defined by a set of outdoor elements of different kinds (model level M2). The goal is to find the right set and constellation of racks, frames, and modules, such that each element is connected to exactly one module. Of course, we aim for a minimal set of hardware. Additionally, various types of constraints restrict the allowed constellations. Typical examples of such constraints are: Some types of models should not be mixed within a frame. Certain types of modules must not be mounted on neighbouring places in the frame.

It shall be noted that for a concrete problem instance on model level M2, it is not known beforehand how many racks, frames, and modules are needed for a valid solution on model level M1. This is why such kinds of problems are called *dynamic* problems in contrast to *static* problems.

3 Approach

According to Fig. 1, inputs to the reconfiguration solver are the problem descriptions M3' and M2', and the legacy configuration M1'. Output is M1" which should be consistent w.r.t. M3' and M2' and as close as possible to M1'.

The basic of idea of our reconfiguration approach is to influence a solver to search in the neighbourhood of the legacy configuration. This is achieved by defining a heuristic for the solver to choose for every decision the same value that has been used in the legacy configuration, whenever possible. In a way this reconstruct the search tree which was built creating the legacy configuration. Deviations from that search tree should only happen when previously consistent choices are inconsistent now.

Our approach will perform poorly, if there is no consistent configuration close to an inconsistent legacy configuration. But the approach will perform well if only a small percentage of the overall configuration is modified during reconfiguarion, which is the case in most reconfiguration scenarios in practice especially for large configurations. E.g., from our experience in the rail automation domain, most system modifications are only very local changes in the outdoor installation.

To show that our approach is applicable to different solving paradigms we implemented it in two standard AI technologies, answer set programming (ASP) and constraint satisfaction (CSP).

For the ASP implementation we used Potassco's `clingo` which allows to define domain specific heuristics within ASP with a special predicate `_heuristic`. With this predicate the solver can be influenced to prefer certain atoms during solving. By giving the legacy configuration as preferred heuristic facts we achieve a kind of replay of the original configuration. Details are described in Chap. 3.1.

CSP systems often are more open to adaptations of the search procedure than ASP solvers. For our experiments, we used Choco which allows to plug in your own variable and value ordering heuristics used during backtrack search. We wrote variable and value ordering heuristics which prefer the decisions of the legacy configuration. Details are described in Chap. 3.2.

We chose Potassco and Choco for our experimental implementations because they are well recognized implementations of ASP

and CSP technology. Potassco is very fast and regularly wins competitions. Choco is a standard constraint solver in Java. Most likely there are CSP systems with better performance, but we belief that all solvers based on a backtracking scheme suffer from the same fundamental behaviour of sometimes running into heavy backtracking as shown in our evaluation (cf. Sec. 4).

3.1 Answer set programming

ASP is a declarative problem solving approach, which originated in the area of knowledge representation and reasoning [10]. To implement our approach we used the Potassco ASP implementation [12] and our OOASP framework [16]. OOASP provides special predicates for describing object-oriented knowledge bases in ASP. Our running example can be declared in OOASP as follows:

```
ooasp_class("hw","Rack").
ooasp_class("hw","Frame").
ooasp_class("hw","Module").
ooasp_class("hw","ModuleA").
ooasp_class("hw","ModuleB").
ooasp_class("hw","Element").
ooasp_class("hw","Track").
ooasp_class("hw","Signal").

ooasp_subclass("hw","ModuleA","Module").
ooasp_subclass("hw","ModuleB","Module").
ooasp_subclass("hw","Track","Element").
ooasp_subclass("hw","Signal","Element").

ooasp_assoc("hw","Frame_modules",
    "Frame",1,1,
    "Module",0,5).

ooasp_attribute("hw","Module",
  "position","integer").
ooasp_attribute_minInclusive("hw","Module",
  "position",1).
ooasp_attribute_maxInclusive("hw","Module",
  "position",5).

ooasp_assoc("hw","Module_element",
  "Module",1,1,
  "Element",1,1).
ooasp_assoc("hw","Rack_frames",
  "Rack",1,1,
  "Frame",0,4).
```

In a similar manner the predicates $ooasp\_isa$, $ooasp\_attribute\_value$ and $ooasp\_associated$ are used to define (partial) configurations i.e. instantiations of the object-model.

One standard reasoning task of OOASP is to complete a partial configuration i.e. to add components to the partial configuration until all constraints of the knowledge base are satisfied. For example given a partial configuration consisting only of one track (represented by the fact $ooasp\_isa("c","Track","A1")$), completing a configuration will return all configurations containing one track, with at least one module of type A, one frame and one rack.

3.1.1 Heuristic reconfiguration

Given an inconsistent legacy configuration the default reconfiguration reasoning task in OOASP finds a valid configuration that is cheapest in terms of some user defined cost function. The costs can be either domain-specific or cardinality based. The cardinality based cost function simply counts the difference (in number of facts) of the legacy configuration and the reconfigured configuration. This default reconfiguration task in OOASP is implemented using ASP optimization statements. Unfortunately it has a bad performance for large problem sizes due to the large number of possible configurations. This was one of the motivations for developing a novel approach to reconfiguration with OOASP based on heuristics.

Heuristic reconfiguration for OOASP described in this paper uses the special predicates $\_heuristic(ATOM, TRUTHVALUE, LEVEL)$ from [11] to express heuristics in ASP . If during search $\_heuristic(ATOM, TRUTHVALUE, LEVEL)$ can be derived and LEVEL > 0 then the ASP solver will prefer setting atom ATOM to TRUTHVALUE, if given a choice. With the heuristic predicates the order in which solutions (answer sets) are found can be influenced. It does not affect the set of found solutions.

To implement our heuristic approach we add the facts describing the legacy configuration as heuristics facts $\_heuristic(FACTFROMLEGACY, true, 1)$ to the ASP program and run the default OOASP configuration task with this heuristic information. This way the ASP solver is expected to find configurations that are close (cardinality based) to the legacy configuration first.

For example given the heuristic information below the ASP solver will try to assign track A1 first to the module M1 and set its position to 4.

```
% user supplied fact
ooasp_isa("c","Track","A1")
% legacy configuration converted to heuristic
_heuristic(
  ooasp_isa("c","ModuleA","M1"),
  true,1).
_heuristic(
  ooasp_attribute_value(
    "c",
    "position",
    "M1",4),
  true,1).
_heuristic(
  ooasp_associated(
    "c",
    "Module_element",
    "M1","A1"),
  true,1).
...
```

3.2 Constraint satisfaction

The encoding of a dynamic problem with a standard constraint formalism (like MiniZinc[2] or Choco[3]) makes it necessary to define a maximum number of instances of each object type. If, e.g., we allow at most 5 racks for our example problem in Section 2, the maximum numbers of instances for the other types can be computed by cardinality propagation: 20 frames, 320 modules and 320 elements. For complex networks of classes, this cardinality propagation is not trivial [4].

We briefly sketch here the CSP encoding of configuration problems we used in our implementation. We use a pseudo code notation,

[2] http://www.minizinc.org/
[3] http://choco-solver.org/

which could directly be translated to a concrete CSP notation (e.g. Choco). To represent the instances of a class, we use an array of boolean variables representing which element is actually used in a solution and which not. E.g., let r be that variable array for the class `Rack`. nr be the maximum number of racks.

$$r_i \in \{0,1\}, \quad \forall_i \in \{1,...,nr\}$$

The following symmetry breaking constraints states that unused instances are always in the rear part of the array:

$$r_i = 0 \rightarrow r_j = 0, \quad \forall_{i,j} \in \{1,...,nr\}, i < j$$

Attribute encoding is straight-forward. For each possible instance of a class, a CSP attribute variable is used. E.g., attribute `modulePos` of modules is represented by the variable array mp (let nm be the maximum number of modules):

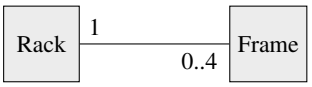
$$mp_i \in \{-1,1,...,16\}, \quad \forall_i \in \{1,...,nm\}$$

We provide a special attribute value (-1 for mp) for attributes of unused components. The following constraint states that unused components must have this special value, and used components must have a value from the regular domain of the attribute.

$$m_i = 0 \leftrightarrow mp_i = -1, \quad \forall_i \in \{1,...,nm\}$$

The interesting part of the model is the encoding of associations. A very general approach is to represent each association by a matrix of boolean variables on the instances of the two involved classes. A matrix entry $m_{ij} = 1$ means that object with index i is associated to object with index j. This representation needs a lot of CSP variables and makes the formulation of consistency constraints on associations quite intricate, causing low solving performance. Another approach [7] models association links as ports; an object has n association variables, where n is the maximum cardinality of the association.

We use a simpler representation: For a 1:n-association, we use a variable array on the n side of the association. Each such variable points to the associated object, or has value -1, if not connected. Example: The association



is represented as an integer variable fr for each frame:

$$fr_i \in \{-1,1,...,nr\}, \quad \forall_i \in \{1,...,nf\}$$

The special value -1 is used if a frame is not associated to a rack at all. This special value is also used for unused frames.

$$f_i = 0 \rightarrow fr_i = -1, \quad \forall_i \in \{1,...,nf\}$$

Additional consistency constraints are needed to rule out invalid association constellations. Each used frame must be connected to a rack:

$$f_i = 1 \rightarrow fr_i \in \{1,...,nr\}, \quad \forall_i \in \{1,...,nf\}$$

Frames can only be connected to used racks:

$$fr_i \geq 1 \rightarrow r_{fr_i} = 1, \quad \forall_i \in \{1,...,nf\}$$

Only up to 4 frames are allowed in a rack:

$$|\,\{fr_j \mid j \in \{1,...,nf\}, fr_j = i\}\,| \leq 4, \quad \forall_i \in \{1,...,nr\}$$

Example for a constraint on attributes and associations: All modules in a frame must have different module positions:

$$mf_i = mf_j \wedge mf_i \geq 1 \rightarrow mp_i \neq mp_j, \quad \forall_{i,j} \in \{1,...,nm\}, i < j$$

It should be noted that such object-constraint mapping on dynamic problems (where the number of instances in a solution is not known beforehand) has many disadvantages: (1) The representation of objects as a flat set of constraint variables is very unnatural and hard to read, debug, and maintain. This can be mitigated by an automatic translation from objects to constraints. (2) A maximal set of possible object instances must already be provided at problem formulation. Decisions on maximum values are in general not easy; too few objects could rule out possible solutions; too many objects blows up the problem size. (3) Current constraint solvers (mainly based on backtracking) are very sensitive to changes. Small changes in the variable structure or of the constraints could hugely influence solving performance, which makes a repeated tuning of the variable and value ordering heuristics necessary. (4) The representation of associations is crucial. A simple representation, as described above, does not directly support n:m associations and ordered associations. More elaborate encodings are difficult to handle in terms of constraint formulations, and often impair performance. (5) Modelling of inheritance additionally increases representation complexity. (6) Attributes of more complex types, like reals, multi-valued variables, or strings, are often not supported at all in constraint systems.

To formalize *reconfiguration* in terms of standard CSP, we first need to define a metric to have a notion of the distance between a legacy configuration and a reconfigured solution. Let (V, D, C) be a CSP with variables V, their domains D, and constraints C. An assignment is a tuple (v, d), $v \in V$, $d \in D_v$, representing the assignment of value d to variable v. Let A be a set of assignments. $vars(A)$ is the set of variables in A: $vars(A) = \{v \mid (v,d) \in A\}$. $vars(A1, A2)$ is the set of variables both in $A1$ and $A2$: $vars(A1, A2) = \{v \mid v \in vars(A1), v \in vars(A2)\}$. $\mathrm{diff}(A1, A2)$ is the number of assignments with different values on the common variables in $A1$ and $A2$:

$$\mathrm{diff}(A1, A2) = |\,\{v \mid v \in vars(A1, A2), \\ (v, d1) \in A1, (v, d2) \in A2, d1 \neq d2\}\,|$$

diff defines a simple metric on the space of all assignment sets of a CSP. The CSP reconfiguration problem can now be simply defined as follows: Let (V, D, C) be a CSP. Let A be an assignment on V or a subset on V. A is potentially inconsistent w.r.t. the constraints C. A *reconfiguration* A' is a consistent assignment on the variables V (i.e., A' is a solution of the corresponding CSP) which minimizes $\mathrm{diff}(A, A')$.

Reconfiguration can be simply implemented by slightly changing the backtracking search procedure. We can't start with the legacy configuration (a variable assignment), because it is potentially inconsistent and standard backtracking would stop with output `failure` immediately. But we could solve the problem from scratch and use the legacy configuration to guide expanding the search tree. Each time when a value for the current variable is selected, the value of the legacy configuration - if an assignment tuple for that variable is

part of the legacy configuration - is preferably chosen. Of course, if that value is inconsistent with the current constellation, an alternative value is taken. But basically, the legacy configuration is replayed, and changes are made only because of inconsistent states. The result is a consistent configuration which is very similar to the legacy configuration, which is exactly what we want - we want minimal reconstruction of the system in the field.

A brief note on our implementation: We used the constraint solver Choco V3.2.2 to represent and solve configuration problems (as described in the first part of this subsection) and replay-based reconfiguration. Choco allows to plug in your own value selector by implementing the interface `IntValueSelector`. With only a few lines of code, we extended the standard value selector of Choco by preferring the values stored in a given legacy configuration. If a legacy value for the current variable is not available or inconsistent, standard Choco value selection is used.

Advantages: (1) This method is light-weight, i.e., no additional modelling concepts (like cost functions) are needed. Changes in the existing backtracking search procedures are minimal. (2) Most of the existing backtracking algorithms (intelligent backtracking, etc.) and variable and value ordering heuristics can still be used with only minimal adaptations. (3) Replay-based reconfiguration can be applied to inconsistent configuration fragments, which is not the case for standard backtracking and consistency algorithms. (4) The method is complete (a solution is found, if one exists).

Disadvantages: (1) It is not guaranteed, that a solution with minimal changes is found. The quality of the solution depends on the variable/value ordering heuristics used. Nevertheless, results of our prototypical implementation have shown, that the reconfiguration solutions are often the optimum or very close to the optimum. (2) Replaying an inconsistent configuration may lead search into inconsistent branches which may heavily impair performance. (3) The approach is suited only for domains where a cardinality based cost function is applicable, e.g. homogeneous hardware configuration problems.

4 EVALUATION

We did experimental evaluations on our ASP (cf. Section 3.1) and CSP (cf. Section 3.2) implementations using randomly generated problem instances for the example problem domain sketched in Fig. 2. We ran the tests on a standard Windows 7 machine with a Intel dual-core i5 CPU and 8 GB RAM. We used clingo V4.4.0 from the Potassco suite for the ASP implementation, and Choco V3.2.2 for the CSP implementation.

It should be mentioned that we intentionally didn't use a high-performance hardware setting and we did not do any coding optimizations. Of course, ASP and CSP experts could find encodings which would perform better, but we wanted to test if AI techniques like ASP and CSP could be used by engineers with just standard skills in these techniques.

The input values for our HW configuration problem are the number and types of `Elements`. We generated randomly a set of input problem instances, solved them, made random changes to the solutions and applied the heuristic replay-based solvers (both ASP and CSP) to solve the reconfiguration problem. We measured solving time, memory consumption, and quality of the reconfiguration solution (i.e., how close is the result to the original configuration).

It should be mentioned that integration of reasoning functionality into our configuration infrastructure – an object-oriented data model and environment implemented in Java – was easier with Choco than with clingo, because Choco provides a Java API.

4.1 Performance: Time

Figures 3 and 4 show the solving time results for our ASP and CSP implementations. The x-axis shows the different problem sizes in terms of number of `Elements`. Note that the number of objects in a configuration solution is far higher than these values, because all the HW elements (racks, frames, modules) and internal variables are created during solving. The y-axis shows the solving execution time in seconds. For ASP, this is the sum of grounding and SAT solving time.

We incremented the problem size, i.e. the number of `Elements`, in steps of 10. We generated 40 different configurations per problem size, modified them randomly and solved the reconfiguration problem. Black circles in the plots are the measured times for each run. Blue squares are the mean runtime values for that problem size. Red triangles indicate timeouts (time $>$ 2 minutes).

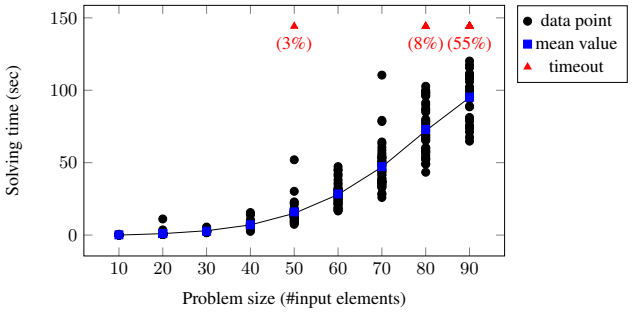


Figure 3. ASP solving times.

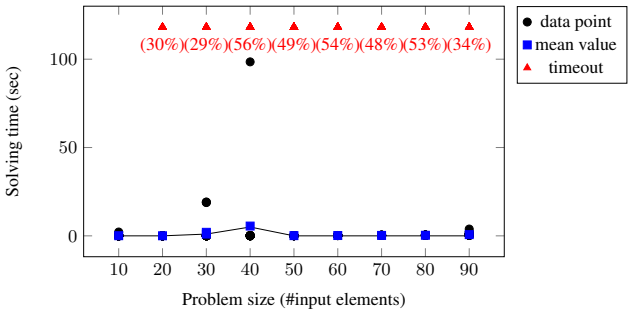


Figure 4. CSP solving times.

We made the following interesting observations:

- ASP is much more robust than CSP. For problems up to a size of ca. 90 `Elements` ASP finds a solution in a well predictable time. In contrast, CSP often needs a lot of time even for small problems and very often runs into timeout.
- Consequently, the sizes of problems where ASP finds a solution in acceptable time is also well predictable. In our test environment, ca. 100 `Elements` are the upper limit for ASP.
- CSP is much more sensitive to the input problem constellation. If the backtracking procedure makes invalid choices in the first part

of the search tree, backtracking gets out of hand. This is why CSP runs very often into timeout even for small problem sizes. In cases without or with little backtracking, CSP is very fast, even for large problems.
If one is willing to tune the variable and value ordering heuristics for her/his specific problem instance, CSP can solve much bigger problems than ASP very efficiently. The key is to avoid backtracking.

- The variance of runtime continuously grows with problem size for ASP. This is not the case for CSP. If CSP manages to solve the problem, it can do it most of the time very quickly. For the solvable problem sizes, there is rarely a difference in the CSP runtimes depending on the size of the input variables.

4.2 Performance: Memory

For all the test cases, we also measured indicators for memory consumption (cf. Fig. 5). For ASP, we used grounding size in MBytes. For CSP, we counted the number of CSP variables used. Note that, aside from user-defined variables (cf. Section 3.2), Choco creates a lot of additional, internal variables. Of course, grounding memory size for ASP and numbers of variables in CSP cannot be compared directly, but they give good indicators about the memory growth rate depending on the input configuration size.

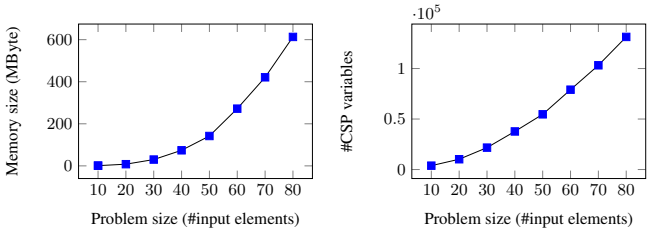


Figure 5. (a) ASP grounding memory size. (b) CSP number of variables.

Not surprisingly, memory of both ASP and CSP grows with accelerated speed depending on the problem size. ASP grows with a slightly higher rate. Not only in the context of reconfiguration, ASP often shows its limits at grounding. Most of the execution time and a big amount of memory is used for grounding.

To give estimations of consumed memory for CSP is a bit more subtle. As shown in Fig. 5(b), we used the number of variables as memory indicator. A rough memory profile using Choco's statistics functionality has shown, that for 100,000 variables ca. 20 MByte RAM is consumed (for the cases without heavy backtracking). This means that CSP's footprint is roughly 20 times smaller than ASP's footprint.

4.3 Performance: Quality

To evaluate the quality of a reconfiguration we measured the distance of the original, legacy configuration to the reconfigured solution. We used a graph-based difference metric counting the differences in the rack/frame/module constellation of the legacy configuration to the reconfiguration.

The first and simplest case is to provide a legacy configuration which is already consistent. This means that a reconfiguration should reproduce the legacy configuration without any changes. Both ASP and CSP did this in many test cases of various sizes.

The more interesting case is a legacy configuration which is inconsistent to the problem description. For each problem size (starting from 10 `Elements` up to 80 `Elements` in steps of 10), we randomly modified up to 20% of a valid configuration. For each problem size, we did 40 different tests.

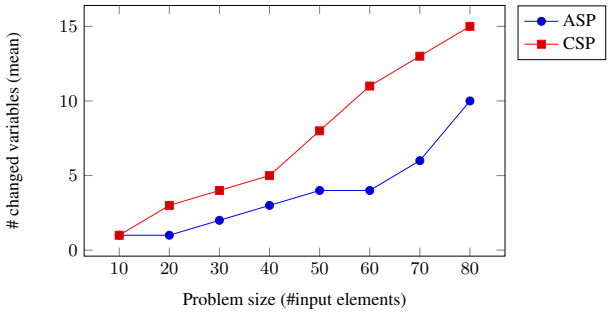


Figure 6. ASP and CSP reconfiguration quality.

The results are shown in Fig. 6. ASP most of the time finds solutions of high quality. In fact, for smaller problem sizes we could manually verify that ASP nearly all the times finds the optimal solution. With CSP, the mean distance to the legacy configuration is a bit higher than with ASP, but has an acceptable quality on most of the cases.

4.4 Evaluation Summary

Table 2 gives a summarized comparison of our ASP and CSP reconfiguration encodings and the results of our experimental evaluations. For solving placement problems like our hardware example, there is no clear winner. If the problem is of moderate size, ASP provides a sound, predictable and easy-to-use reasoning functionality. For larger problem, CSP may be better, but probably additional coding is needed for tuning search.

5 RELATED WORKS

Related to our work presented in this paper are all techniques for finding a valid reconfiguration for a given, possibly inconsistent configuration (fragment). The main research approaches are:

Repair-based approaches. Repair-based approaches aim for finding diagnoses and repairs to conflicting requirements or constraints [5]. Usually, methods from model-based diagnosis are used (e.g., minimal hitting sets). Repair-based approaches are mainly studied in the context of recommender systems [6]. These approaches are complete, they are based on a clean, formal theory, and they typically take user needs into account. Those repairs are in favour which may be of most usefulness for the user. When applied in a configuration context based on consistency and search algorithms, repair-based methods introduce additional reasoning techniques which must be integrated into the configurator framework. Our heuristic, replay-based approach uses conventional solving techniques with slight modifications for reconfiguration.

Minimization of modification costs. The basis of these approaches is the definition of cost functions for the different modification operations [9]. Reconfiguration is then finding a valid modification of the configuration which minimizes the sum of all modification costs. Such techniques have been, e.g., intensively studied in the research project RECONCILE[4]. The possibility of defining elaborate cost functions for configuration modifications along with a complete optimization search procedure (in [9], based on ASP) is a great advantage for applications where modifications in the field are expensive. But this comes at the price of considerable additional modelling concepts for cost functions and often declined solving performance. Compared to that, our approach is light-weight in the sense that no additional modelling is necessary, and most of the advanced backtracking algorithms with only minimal adaptations are applicable.

Table 2. Comparison of ASP and CSP reconfiguration modelling and behaviour.

	ASP	CSP
Robustness	High Predictable	Low Very sensitive to input constellation and problem formulation
Performance	Good for small problem sizes Does not scale for larger problems	Very good, if no or little backtracking, else timeout Probability of timeout grows with problem size
Memory footprint	High (grounding!)	Low
Solution quality	Very good Most of the time the optimum or close to the optimum	Never better than ASP, but most of the time acceptable
Integrability	Typically, ASP systems are not as easy to integrate into a Java environment as CSP systems. The ASP solvers Potassco[5] and Smodels[6] provide C++ libraries, DLV[7] recently provided a Java interface (JDLV[8]).	Many CSP solvers support APIs to programming languages like Java (e.g. Choco[9], JaCoP[10]) or C++ (e.g. Gecode[11], Minion[12]).
Problem encoding	Favoured is an automatic transformation from the object-oriented problem description to ASP. Direct ASP encoding is also possible, because ASP has a compact syntax and is readable.	Direct encoding of an object-oriented data model with a CSP system is quite intricate and error-prone. Automatic transformation is highly recommended.

Local search methods. The main alternatives to backtracking-like search are so-called heuristic (or local) search strategies which try to find solutions in a hill-climbing manner (e.g. greedy search algorithms, genetic algorithms). In the context of product configuration, Generative CSP (GCSP) is an extension of standard CSP and has been introduced in [8] for large, dynamic configuration problems. In GCSP, mainly local search techniques - *repair-based local search* - are used because no fast complete search methods are available yet for dynamic systems. Repair-based local search tries to find local modifications of an inconsistent or incomplete configuration. Thus, this technique intrinsically can deal with inconsistent configuration (fragments). Complex, dynamic problems can be modelled in a very natural way using object-oriented concepts. The main disadvantage of local search methods is that they are incomplete – they may get stuck in a local optimum during search and may not find a solution, even if one exists. Compared to that, our approach is complete, because it is based on an exhaustive tree search (backtracking).

Rule-based approaches. Especially in model-driven engineering, a lot of research in model synchronization has been done and is still on-going. Correspondences between two models are defined as transformation rules, describing how values from one model are mapped to another model. Examples of such systems are triple graph grammars [13] or JTL [2]. Model synchronization (corresponding to reconfiguration in our definition) is done by triggering the transformation rules. Applying such methods to a reconfiguration problem in product configuration means that all necessary types of modification operations for transforming an invalid configuration to a valid one must be specified explicitly. Our heuristic, replay-based approach does not need any additional knowledge like transformation rules. Reconfiguration is guided by a legacy configuration and a declarative problem specification (models M3 and M2 in Tab. 1).

Common to all these approaches – at least to a certain degree – is that reconfiguration actions are modelled on a declarative level. The specification of potential modification operations and reconfiguration reasoning are separated. Another approach used in industry (e.g. in factory facilities, steel plants) is to offer upgrade or modernization packages. There the focus lies on finding and recommending modernization packages which are appropriate to add functionalities to a system in the field.

6 CONCLUSION AND FUTURE WORK

There is no standard way of doing reconfiguration for product configuration especially for large problem sizes. In this paper we showed the implementation of a heuristic approach to reconfiguration using standard solving techniques and the applicability up to moderate problem sizes. The main challenge for using these techniques in an industrial setting are grounding size and solving time. Grounding size typically can be influenced by finding a better encoding or by problem decomposition. Solving time is also influences by the encoding of the problem and by finding the right heuristic for the domain.

Because of the heterogeneous nature of the constraints in product configuration coming up with a good encoding and heuristics for a problem is currently as much an art as a science and requires an experienced knowledge engineer. Also it requires experiments with different solving paradigms as SAT, ASP, CSP, MIP etc. Therefore we welcome the further integration of AI and OR solving techniques that have taken place in the last years as we believe there will not be THE solving technique for product (re-)configuration in the foreseeable future.

For the future we plan to further study and improve heuristic reconfiguration solving techniques and to apply them to fields beyond

[4] http://isbi.aau.at/reconcile/
[5] http://potassco.sourceforge.net/
[6] http://www.tcs.hut.fi/Software/smodels/
[7] http://www.dlvsystem.com/
[8] http://www.dlvsystem.com/jdlv/
[9] http://choco-solver.org/
[10] http://www.jacop.eu/
[11] http://www.gecode.org/
[12] http://minion.sourceforge.net/

product configuration. As we have seen in our experiments, CSP solving – though it is very fast and produces good results if it doesn't fall into a heavy backtracking trap – currently isn't robust enough to be applied in an industrial environment. We believe that the integration of additional heuristics which are automatically derived from the problem domain or techniques like lazy clause generation [17] will fix this problem of poor robustness. Also the integration of CSP and ASP (CASP [1]) looks promising.

Interesting fields beyond product configuration, where reconfiguration methods could be applied, are:

- *Production configuration:* With the increasing demand for individualized products, the need for flexible production processes, modular factories and intelligent production infrastructures is also increasing. Factories of the future are generic production facilities, that can be easily adapted to the needs of the product to be manufactured [3]. This means, before the factory can manufacture products of a product line, it has to be physically reconfigured for the specific production setting. This includes reconfiguration of the cyber-physical components of the factory [15], and therefore the need for fast, flexible and robust reconfiguration technologies.
- *Model synchronization:* In model-driven engineering, model synchronization is the task of finding a mapping between overlapping concepts of two different models. Typically, the overlaps of the two models are described as a correspondence model, including constraints which define the dependencies and interactions between the models. This situation can be seen as a reconfiguration problem: Given are two models (e.g., configuration instances of two different configuators) which have been changed in the course of a new system version, and a correspondence model. The reconfiguration problem is now to find changes in the two evolved models which are (a) consistent to their domain model, (b) consistent to the correspondence model, and (c) as close as possible to the original models.
- *Case-based reasoning:* In case-based reasoning, a database of solutions from previous problems are used to find a solution to a new problem [18]. Usually, no perfectly fitting solution could be found, but one which solved a similar problem. We think that our heuristic, replay-based reconfiguration procedures could be applied to the reuse/revise phase of case-based reasoning: To solve a new problem try to rebuild the configuration of a solved problem.

ACKNOWLEDGEMENTS

This work has been funded by the Vienna Business Agency in the programme ZIT13-plus within the project COSIMO (Collaborative Configuration Systems Integration and Modeling) under grant number 967327, and by FFG FIT-IT within the project HINT (Heutistic Intelligence) under grant number 840242.

REFERENCES

[1] Marcello Balduccini and Yuliya Lierler, 'Integration schemas for constraint answer set programming: a case study', *TPLP*, **13**(4-5-Online-Supplement), (2013).

[2] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio, 'Jtl: A bidirectional and change propagating transformation language', in *Software Language Engineering*, eds., Brian Malloy, Steffen Staab, and Mark van den Brand, volume 6563 of *Lecture Notes in Computer Science*, 183–202, Springer Berlin Heidelberg, (2011).

[3] D. Dhungana, A. Falkner, A. Haselböck, and H. Schreiner, 'Smart factory product lines: A configuration perspective on smart production ecosystems', in *Manuscript submitted for publication*, (2015).

[4] Ingo Feinerer and Gernot Salzer, 'Numeric semantics of class diagrams with multiplicity and uniqueness constraints', *Software and System Modeling*, **13**(3), 1167–1187, (2014).

[5] Alexander Felfernig, Gerhard Friedrich, Monika Schubert, Monika Mandl, Markus Mairitsch, and Erich Teppan, 'Plausible repairs for inconsistent requirements', in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pp. 791–796, (2009).

[6] Alexander Felfernig, Erich Teppan, Gerhard Friedrich, and Klaus Isak, 'Intelligent debugging and repair of utility constraint sets in knowledge-based recommender applications', in *Proceedings of the 2008 International Conference on Intelligent User Interfaces, January 13-16, 2008, Gran Canaria, Canary Islands, Spain*, pp. 217–226, (2008).

[7] Adel Ferdjoukh, Anne-Elisabeth Baert, Annie Chateau, Remi Coletta, and Clémentine Nebut, 'A CSP approach for metamodel instantiation', in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013*, pp. 1044–1051, (2013).

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

[9] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, '(re)configuration based on model generation', in *LoCoCo*, pp. 26–35, (2011).

[10] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012.

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

[12] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider, 'Potassco: The potsdam answer set solving collection', *AI Commun.*, **24**(2), 107–124, (2011).

[13] Frank Hermann, Hartmut Ehrig, Claudia Ermel, and Fernando Orejas, 'Concurrent model synchronization with conflict resolution based on triple graph grammars', in *Fundamental Approaches to Software Engineering*, eds., Juan de Lara and Andrea Zisman, volume 7212 of *Lecture Notes in Computer Science*, 178–193, Springer Berlin Heidelberg, (2012).

[14] U. Junker, 'Configuration', in *Handbook of Constraint Programming*, eds., F. Rossi, P. vanBeek, and T. Walsh, pp. 837–873. Elsevier, (2006).

[15] Kunming Nie, Tao Yue, Shaukat Ali, Li Zhang, and Zhiqiang Fan, 'Constraints: The core of supporting automated product configuration of cyber-physical systems', in *Model-Driven Engineering Languages and Systems*, eds., Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter Clarke, volume 8107 of *Lecture Notes in Computer Science*, 370–387, Springer Berlin Heidelberg, (2013).

[16] Gottfried Schenner, Andreas Falkner, Anna Ryabokon, and Gerhard Friedrich, 'Solving object-oriented configuration scenarios with asp.', *Proceedings of the 15th International Configuration Workshop*, 55–62, (2013).

[17] Peter J. Stuckey, 'Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving', in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, pp. 5–9, (2010).

[18] Hwai-En Tseng, Chien-Chen Chang, and Shu-Hsuan Chang, 'Applying case-based reasoning for product configuration in mass customization environments', *Expert Syst. Appl.*, **29**(4), 913–925, (2005).