ReMax – A MaxSAT aided Product (Re-)Configurator

Rouven Walter and **Wolfgang Küchlin**[1]

Abstract. We introduce a product configurator with the ability of optimal re-configuration built on MaxSAT as the background engine. A product configurator supported by a SAT solver can provide an answer at any time about which components are selectable and which are not. But if a user wants to select a component which has already been disabled, a purely SAT based configurator does not support a guided re-configuration process. With MaxSAT we can compute the minimal number of changes of component selections to enable the desired component again. We implemented a product configurator — called ReMax — using state-of-the-art MaxSAT algorithms. Besides the demonstration of handmade examples, we also evaluate the performance of our configurator on problem instances based on real configuration data of the automotive industry.

1 Introduction

Using Propositional Logic encodings and SAT solving techniques to answer the question whether a formula is satisfiable or not has a wide range of applications [10]. The application of SAT solving for verification of automotive product documentation for inconsistencies, e.g. within the bill-of-materials, has been pioneered by Küchlin and Sinz [7].

In [16] we considered applications of MaxSAT in automotive configuration. We mentioned the possible usage of re-configuration with MaxSAT to make an invalid configuration valid again by keeping the maximal number of the customer selections. Re-configuration is of highly practical relevance [9]. For example, the after-sales business in the automotive industry wants to extend, replace or remove components with minimal effort while keeping the configuration valid.

In this paper, we extend this idea by considering product configuration in general. We focus on product configuration based on families of options, because this is the normal case when a user configures a product. Within a family of options, the user must select exactly one option out of a *regular family* or else may select at most one option out of an *optional family*. With the focus on families, we can distinguish two solving approaches:

1. **SAT Solving**: With a SAT aided product configurator, we can validate a configuration after each step of the configuration process.
2. **MaxSAT Solving**: With a MaxSAT aided product configurator, we can compute an optimal solution for an invalid configuration, such that a user has to make a minimal number of changes in the current configuration to regain validity.

We identify different use cases. We describe them in detail and make remarks about extensions or variants of them. We also show how a user process can look like using a MaxSAT aided product configurator.

This paper is organized as follows. Section 2 introduces all relevant mathematical definitions and notations needed for the later sections. In Section 3 we describe the basic concepts of SAT-based product configuration. Section 4 shows use cases of SAT aided product configuration. After that we describe use cases for MaxSAT aided product configuration in detail in Section 5 and illustrate a possible configuration process. Sections 6 and 7 describe the techniques we used for our implementation and experimental results with benchmarks based on industrial configuration instances. Section 8 describes related work and finally, Section 9 concludes this paper.

2 Preliminaries

We consider propositional formulas with the standard logical operators $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ over the set of Boolean variables X and with the constants $\perp$ and $\top$, representing false and true, respectively. Let $\mathrm{vars}(\varphi)$ be the set of variables of a formula φ. We call a formula φ *satisfiable*, if there exists an *assignment*, a mapping from the set of Boolean variables X to $\{0, 1\}$, under which the formula φ evaluates to 1. The evaluation procedure is assumed to be the standard evaluation for propositional formulas. The Boolean values 0 and 1 are also referred to as *false* and *true*. If no such assignment exists, we say the formula is *unsatisfiable*. The question whether a propositional formula is satisfiable or not is well-known as the *satisfiability (SAT) problem*, which is NP-complete.

In most cases a SAT solver accepts only formulas in *conjunctive normal form* (CNF). A formula in CNF is a conjunction of *clauses*, where a clause is a disjunction of *literals* (variables or negated variables). Let $\mathrm{var}(l)$ be the variable of a literal l.

If a formula $\varphi = \bigwedge_{i=1}^{k} \bigvee_{j=1}^{m_i} l_{i,j}$ in CNF is unsatisfiable, we can ask the question about the maximal number of clauses that can be satisfied at the same time. This optimization variant of the SAT problem is called *maximum satisfiability (MaxSAT) problem*. The corresponding question about the minimal number of unsatisfied clauses is analogously called *minimum unsatisfiabitlity (MinUNSAT) problem*. A solution to one of the two problems can be used to easily compute the solution of the other one, because the following relationship holds: $k = \mathrm{MaxSAT}(\varphi) + \mathrm{MinUNSAT}(\varphi)$. It is worth noting that a model of the optimum of the MaxSAT problem is also a model of the optimum of the MinUNSAT problem and vice versa. In general, there are several models for the optimum.

The MaxSAT problem can be extended in different ways: (i) we can assign a non-negative integer weight to each clause (denoted with (C, w) for a clause C and a weight w) and ask for the maximum sum of weights of satisfied clauses, which is known as the *Weighted MaxSAT problem*, (ii) we can split the clauses in *hard* and *soft* clauses and ask for the maximum number of satisfied soft clauses while sat-

[1] Symbolic Computation Group, WSI Informatics, Universität Tübingen, Germany, `www-sr.informatik.uni-tuebingen.de`, email: `{walterr,kuechlin}@informatik.uni-tuebingen.de`

isfying all hard clauses, which is known as the *Partial MaxSAT problem*, and finally (iii) we can combine both specifications, which is known as the *Weighted Partial MaxSAT problem*. The mentioned relationship above between the MaxSAT and MinUNSAT problem also holds for all MaxSAT variants.

Given a set of Boolean variables $\mathrm{F} = \{\mathrm{M_1}, ..., \mathrm{M_n}\}$ and the restriction that *exactly one* variable has to be satisfied, $\sum_{i=1}^{n} M_i = 1$, we call the set F a *regular family* and the elements *members* of the family. For example, given a set of Boolean variables $\mathrm{E} = \{\mathrm{E_1}, \mathrm{E_2}, \mathrm{E_3}\}$ representing the selectable engines of a car. An engine is chosen if and only if the corresponding variable is set to true. A car has exactly one engine, which makes the set E a family.

Given a family $\mathrm{F} = \{\mathrm{M_1}, ..., \mathrm{M_n}\}$ with the restriction that *at most one* variable has to be true, we call the set F an *optional family*. For example, given a set of Boolean variables $\mathrm{AC} = \{\mathrm{AC_1}, \mathrm{AC_2}, \mathrm{AC_3}, \mathrm{AC_4}\}$ representing the selectable air conditioners of a car. An air conditioner is an optional feature in a car, but there can be at most one air conditioner. This makes the set AC an optional family.

The restrictions of a regular family or an optional family are special cases of *cardinality constraints*, which restrict the number of satisified variables of a set of Boolean variables to be $\{\leq, <, =, >, \geq\}$ a non-negative integer k. The restriction for a regular family can be encoded in CNF with the following two formulas, while an optional family can be encoded by using only the second formula:

1. At least one satisfied variable: $\bigvee_{i=1}^{n} M_i$
2. At most one satisfied variable: $\bigwedge_{i=1}^{n} \bigwedge_{j=i+1}^{n} (\neg M_i \vee \neg M_j)$

The given encodings for the two special cases $= 1$ and ≤ 1 are very simple and require only $\mathcal{O}\left(n^2\right)$ clauses without adding new auxiliary variables. There are also encodings using auxiliary variables in exchange for a fewer number of clauses [14].

Since we consider only regular and optional family types, more general cardinality constraints than the above-mentioned special cases are not necessary and thus not considered in this paper. In the context of automotive configuration, we usually deal with rules and families of certain model series. For example, the number of seats is fixed and therefore we do not need to handle a family of seats where we would need a cardinality constraint to restrict the selection of seats between two and four.

3 Product Configuration Concepts for SAT-Configuration

In this section, we describe the basic concept of SAT-based product configuration. We concentrate on rules in Propositional Logic, because in our main application context of automotive configuration we always deal with this type of rules. Along with the set of rules we consider families, which results in the following definition for product configuration:

Definition 1. *(Product Configuration Instance[2]) A* product configuration instance *is a triple* $(\mathcal{R}, \mathcal{F}, \mathcal{S})$*:*

- *Set* $\mathcal{R} = \{\varphi_1, \dots, \varphi_k\}$*, where* φ_i *is a propositional formula.*
- *Set* $\mathcal{F} = \{\mathrm{F_1}, \dots, \mathrm{F_m}\}$*, where* $\mathrm{F_i}$ *is a family.*
- *Mapping* $\mathcal{S} : \bigcup_{i=1}^{m} \mathrm{vars}(F_i) \rightarrow \{\mathrm{no}, \mathrm{yes}\} \times (\mathbb{N}_{\geq 0} \cup \{\infty\})$.

[2] In the configuration literature a *product configuration instance* is a solution for a configuration problem, whereas we refer to the term as a description of a product configuration problem.

The following relation holds between rules and families: $\bigcup\limits_{\mathrm{R} \in \mathcal{R}} \mathrm{vars}(\mathrm{R}) \subseteq \bigcup\limits_{\mathrm{F} \in \mathcal{F}} \mathrm{F}$.

The rules $\mathcal{R}$ describe the relationship among the family members of the different families. They determine the possible valid combinations. The set $\mathcal{F}$ contains all optional and regular families. The mapping $\mathcal{S}$ represents the selections and deselections of the family members in respect of a priority. For simplicity reasons we will only use the term selections to refer to both selections and deselections. There are three main cases for a member s:

1. $\mathcal{S}(s) = (c, 0)$ with $c \in \{\mathrm{no}, \mathrm{yes}\}$:
   The user made no decision about the member (priority 0).
2. $\mathcal{S}(s) = (c, p)$ with $c \in \{\mathrm{no}, \mathrm{yes}\}$ and $p \in \mathbb{N}_{\geq 1}$:
   The user made a selection (priority greater zero).
3. $\mathcal{S}(s) = (c, \infty)$ with $c \in \{\mathrm{no}, \mathrm{yes}\}$:
   The user made an indispensable (hard) selection (infinity priority).

We abbreviate the mapping $\mathcal{S}$ for a member s as follows: For a positive selection we write a positive literal s and for a negative selection we write the negative literal $\neg s$. We can then write a single tuple (s, p) with $p \in \mathbb{N}_{\geq 0} \cup \{\infty\}$ and describe the mapping $\mathcal{S}$ as a set of tuples. For simplicity reasons we leave each member s with priority 0 out of $\mathcal{S}$ in the given examples of this paper.

The set $\mathcal{S}$ of selections can be seen as a partial assignment given by the user of the product configurator and can be divided in two disjoint sets of positive and negative selections: $\mathrm{Pos}(\mathcal{S}) := \{(s,p) \mid (s,p) \in \mathcal{S} \text{ and } s \text{ is a positive literal}\}$ and $\mathrm{Neg}(\mathcal{S}) := \{(s,p) \mid (s,p) \in \mathcal{S} \text{ and } s \text{ is a negative literal}\}$.

The priority of a selected member is only relevant when it comes to the question of re-configuration. Then the priorities represent the users preferences.

Example 1. *We consider a product configuration instance* $(\mathcal{R}, \mathcal{F}, \mathcal{S})$*, where* $\mathcal{R}$ *and* $\mathcal{F}$ *describe components of a computer system and dependencies among them. Table 1 shows the families and Table 2 shows the rules. Let* $\mathcal{S} = \emptyset$*, which means a user has not made selections so far.*

Family	Type	Members
M (Mainboard)	regular	$\mathrm{M_1, M_2, M_3, M_4}$
V (Videocard)	regular	$\mathrm{V_1, V_2, V_3, V_4, V_5}$
C (CPU)	regular	$\mathrm{C_1, C_2, C_3, C_4}$
P (Power Supply)	regular	$\mathrm{P_1, P_2}$
CD (CD-Device)	optional	$\mathrm{CD_1, CD_2, CD_3}$
CR (Card-Reader)	optional	$\mathrm{CR_1, CR_2}$

Table 1: Families $\mathcal{F}$ of the computer system

Rules		
M_1	$\rightarrow$	$((\mathrm{V_1} \vee \mathrm{V_2} \vee \mathrm{V_4}) \wedge (\mathrm{C_1} \vee \mathrm{C_3}) \wedge \mathrm{P_1} \wedge \neg\mathrm{CD_1})$
M_2	$\rightarrow$	$((\mathrm{V_2} \vee \mathrm{V_5}) \wedge (\mathrm{C_2} \vee \mathrm{C_3}) \wedge (\mathrm{P_1} \vee \mathrm{P_2}) \wedge \neg\mathrm{CD_1})$
M_3	$\rightarrow$	$((\mathrm{V_3} \vee \mathrm{V_4}) \wedge (\mathrm{C_2} \vee \mathrm{C_3} \vee \mathrm{C_4}) \wedge \mathrm{P_1})$
M_4	$\rightarrow$	$((\mathrm{V_1} \vee \mathrm{V_2}) \wedge (\mathrm{C_1} \vee \mathrm{C_4}) \wedge \mathrm{P_1} \wedge \neg\mathrm{CD_2})$
C_1	$\rightarrow$	$((\mathrm{V_2} \vee \mathrm{V_3}) \wedge \mathrm{P_2})$
C_2	$\rightarrow$	$(\mathrm{V_4} \vee \mathrm{V_5})$
C_3	$\rightarrow$	$(\mathrm{V_3} \vee \mathrm{V_4})$

Table 2: Configuration rules $\mathcal{R}$ of the computer system

We will now define the criteria of a valid configuration:

Definition 2. *(Valid Configuration) A product configuration instance is called a* valid configuration *if the following formula is satisfiable:*

$$\bigwedge_{\mathrm{R}\in\mathcal{R}} R \wedge \bigwedge_{\mathrm{F}\in\mathcal{F}} \mathrm{CC(F)} \wedge \bigwedge_{(s,p)\in\mathcal{S},p\neq 0} s$$

Where $\mathrm{CC(F)}$ *are the appropriate cardinality constraints of a family (described in the preliminaries).*

If a configuration instance is valid, the corresponding (partial) variable assignment (also called model *or* configuration solution*) is of interest, because the variable assignment describes which members are chosen and which are not.*

A configuration solution is in general not complete, e.g. when the selections $\mathcal{S}$ made by a user contain selections with priority 0.

After defining the basic product configuration concepts, we will go into more detail in the next section by describing which use cases of a SAT aided product configurator exist and finally by showing an iterative process of SAT aided product configuration.

4 SAT aided Product Configuration

With SAT solving a product configurator can validate a user's selection and also compute the selectable members for the remaining families. The overall plan is quite simple: Each selection of an option results in a *true* valuation of that option. Regular families result in propagations of the value *false* to the remaining options, after one family member has been selected. Given a partial valuation, it is easy to compute by SAT solving which of the remaining options can still be selected, and which must be set to true or false, respectively, as a consequence of previous selections.

We describe these use cases in detail in the following subsections and afterwards consolidate them in an iterative user process.

4.1 Use Case: Validation & completion of a (partial) selection

Given a product configuration instance $(\mathcal{R},\mathcal{F},\mathcal{S})$, we can validate the selections with a SAT solver by checking the formula of Definition 2 for satisfiability. Algorithm 1 shows the procedure. Only selections with a priority $\neq 0$ are taken into account for the validation.

Algorithm 1: Validation & completion of a (partial) selection

```
Input: $(\mathcal{R},\mathcal{F},\mathcal{S})$
Output: (result, model), where result is true if the (partial)
        selection is valid, otherwise false and model is a
        complete variable assignment
return SAT$\left(\bigwedge_{\mathrm{R}\in\mathcal{R}} \mathrm{CNF}(R) \wedge \bigwedge_{\mathrm{F}\in\mathcal{F}} \mathrm{CC(F)} \wedge \bigwedge_{(s,p)\in\mathcal{S},p\neq 0} s\right)$
```

Because most SAT solvers take CNF as input, we write $\mathrm{CNF}(R)$ to indicate the transformation of an arbitrary rule to its CNF representation. In practice we use a polynomial formula transformation [15, 12] to get an equisatisfiable formula to avoid the potentially exponential blow-up that occurs when using the distributive law.

If the configuration instance is valid, the algorithm also returns a complete variable assignment. This complete variable assignment gives an example which selections have to be made to complete the given configuration instance. In general, the given model is not uniqe and there exist several models.

Example 2. *We reconsider the computer system configuration Example 1. In the following two selection examples, we do not use priorities because we just want to check the validity of the selections.*

1. $\mathcal{S}=\{\mathrm{M}_1,\mathrm{V}_4\}$ *leads to a valid configuration, which can be completed to* $\{\mathrm{M}_1,\mathrm{V}_4,\mathrm{C}_3,\mathrm{P}_1,\mathrm{CD}_3,\mathrm{CR}_2\}$.
2. $\mathcal{S}=\{\mathrm{M}_1,\mathrm{C}_1\}$ *leads to an invalid configuration, because* M_1 *requires* P_1 *and* C_1 *requires* P_2*, but due to the family constraints, both cannot be selected at the same time.*

4.2 Use Case: Computation of selectable members

During the configuration process a user would like to know which of the remaining family members are still selectable, i.e. which selections lead to a valid configuration. We can compute the selectable members by validating the (partial) selections with a SAT solver. Algorithm 2 shows the procedure. We iteratively make one SAT call for each member and check if selecting this member is valid.

Algorithm 2: Computation of selectable members

```
Input: $(\mathcal{R},\mathcal{F},\mathcal{S})$
Output: Mapping $V$ from $\left(\bigcup_{\mathrm{F}\in\mathcal{F}} F\right)$ to $\{\texttt{no},\texttt{yes}\}$ indicating
        whether a member is selectable or not
$V \leftarrow$ Initialize mapping
foreach $m \in \bigcup_{\mathrm{F}\in\mathcal{F}} F$ do
    if
    SAT$\left(\bigwedge_{\mathrm{R}\in\mathcal{R}} \mathrm{CNF}(R) \wedge \bigwedge_{\mathrm{F}\in\mathcal{F}} \mathrm{CC(F)} \wedge \bigwedge_{(s,p)\in\mathcal{S},p\neq 0} s \wedge m\right)$
    then
        $V \leftarrow (m,\texttt{yes})$
    else
        $V \leftarrow (m,\texttt{no})$
return $V$
```

After the computation of selectable members, the SAT aided product configurator can display the result to the user (i.e. by disabling all non-selectable members). Then the user knows about the selectable members.

Remarks:

1. In the special case $\mathcal{S}=\emptyset$, in which no selection has been made by the user so far, the computation of the selectable members implicitly brings up members which can never be part of a valid configuration (*redundant members*) and members which have to be part of each valid configuration (*forced members*).
2. The performance of Algorithm 2 can be improved. If a family already contains a positively selected member, then we know that all remaining members are not selectable anymore due to the family constraints. We just have to check families with no positively selected member.
   The performance can be improved further. We use an incremental and decremental SAT solver, which allows us to load all rules, family constraints and selections first and check each member m by adding and removing the unit clause m from the SAT solver. We do not have to load the invariant constraints repeatedly for each check.

Example 3. *We compute the selectable members for our computer system configuration (see Example 1):*

1. *$\mathcal{S} = \emptyset$: Table 3 shows the remaining selectable members.*

Family	Selectable Memb.	Non-Selectable Memb.
M (Mainboard)	$\mathrm{M}_1, \mathrm{M}_2, \mathrm{M}_3, \mathrm{M}_4$	
V (Videocard)	$\mathrm{V}_1, \mathrm{V}_2, \mathrm{V}_3, \mathrm{V}_4, \mathrm{V}_5$	
C (CPU)	$\mathrm{C}_2, \mathrm{C}_3, \mathrm{C}_4$	C_1
P (Power Supply)	$\mathrm{P}_1, \mathrm{P}_2$	
CD (CD-Device)	$\mathrm{CD}_1, \mathrm{CD}_2, \mathrm{CD}_3$	
CR (Card-Reader)	$\mathrm{CR}_1, \mathrm{CR}_2$	

Table 3: Selectable members for an empty selection

2. *$\mathcal{S} = \{\mathrm{M}_1, \mathrm{V}_4\}$: Table 4 shows the remaining selectable members.*

Family	Selectable Memb.	Non-Selectable Memb.
C (CPU)	C_3	$\mathrm{C}_1, \mathrm{C}_2, \mathrm{C}_4$
P (Power Supply)	P_1	P_2
CD (CD-Device)	$\mathrm{CD}_2, \mathrm{CD}_3$	CD_1
CR (Card-Reader)	$\mathrm{CR}_1, \mathrm{CR}_2$	

Table 4: Result of the selectable members computation

4.3 SAT aided configuration process

Figure 1 illustrates a possible SAT aided configuration process involving both Use Cases 4.1 and 4.2. After the user has made one or multiple selections, the SAT solver validates the current configuration. This results in two cases:

1. **Valid configuration:** In the case of a valid configuration, the user can continue selecting members. Additionally, to guide the user we can compute the selectable members for the current configuration. After new selections, the process iterates.
2. **Invalid configuration:** In the case of an invalid configuration, the user has to take back one or more of the previously made selections. The user can validate each backtracking step again until a valid configuration state is reached.

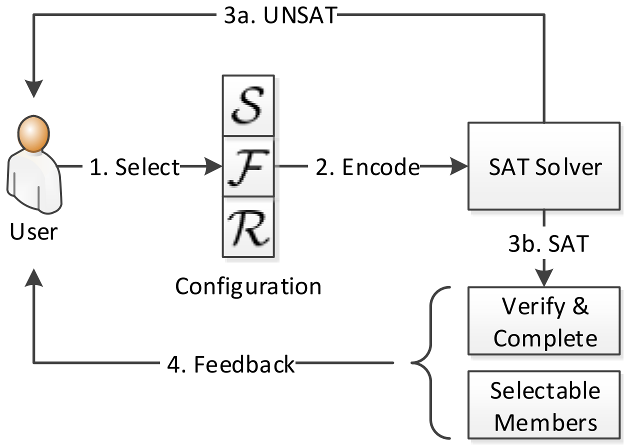


Figure 1. SAT aided configuration process

Remark: If a given complete example model $l_1 \wedge \ldots \wedge l_n$ in the SAT case does not satisfy the demands of the user, she can exclude this model by adding the hard clause $\neg l_1 \vee \ldots \vee \neg l_n$. Then another complete model will be produced if one exists, otherwise we encounter the UNSAT case.

In a SAT aided product configuration process described above, the user is left to herself when it comes to the question which selections should be undone to regain a valid configuration. Perhaps the user made a selection of a highly desired member, which she does not want to take back. Now the user has to try different configuration changes by herself and a guidance is missing which one to choose. This is the point where MaxSAT aided product configuration can help. We will describe re-configuration use cases in detail in the following section.

5 MaxSAT aided Product (Re-)Configuration

In this section we describe how re-configuration can be done with partial (weighted) MaxSAT as a background engine. We show two basic use cases, describe possible variations of them and finally integrate the re-configuration step into our iteractive user process.

5.1 Use Case: Re-configuration of the selections

During the configuration process we may reach a state where we have an invalid configuration. The cause of the conflict can be one or both of the following:

1. The selections $\mathcal{S}$ conflict with the rules $\mathcal{R}$.
2. The selections $\mathcal{S}$ conflict with the family constraints.

We have to re-configure either the rules or the selections to regain validity. For now we consider all rules as hard limitations that we can not soften, which is the common case. We will discuss re-configuration of rules later in Section 5.4.

Considering the rules as a hard restrictions, the question arises, how many of the selections can be kept maximally to reach a valid configuration. Remember, a user may have done multiple selections at once without validating the current configuration and without considering the selectable members. Therefore, removing only the last selection does not lead to a valid configuration again in general. Also the last selection could be of infinity priority, so it is no option for the user to remove the last selection.

To answer the question we set the selections as soft unit clauses and re-configure the selections with a partial MaxSAT solver. The following encoding represents our requirements:

$$\begin{aligned}
\text{Hard} &:= \bigcup_{\mathrm{R} \in \mathcal{R}} \mathrm{CNF}(\mathrm{R}) \cup \bigcup_{\mathrm{F} \in \mathcal{F}} \mathrm{CC}(\mathrm{F}) \cup \bigcup_{(s,p) \in \mathcal{S}, p=\infty} \{s\} \\
\text{Soft} &:= \bigcup_{(s,p) \in \mathcal{S}, p \neq 0, p \neq \infty} \{s\}
\end{aligned}$$

Selections with priority ∞ are also considered as indispensable and will be encoded as hard unit clauses. Only dispensable selections will be re-configured. Algorithm 3 shows the re-configuration procedure.

With the resulting model, we can give the user an example of a complete selection which requires a minimal number of changes in order to regain a valid configuration compared to the original selections. Or, the other way round, the model gives an example about how to keep the maximal number of selections.

Algorithm 3: Re-Configuration of a (partial) selection

```
Input: $(\mathcal{R}, \mathcal{F}, \mathcal{S})$
Output: $(\mathrm{optimum}, \mathrm{model})$, where $\mathrm{optimum}$ is the minimal
        number of changes to regain a valid configuration and
        $\mathrm{model}$ is a model for the optimum
$\mathrm{Hard} \leftarrow \emptyset$
$\mathrm{Soft} \leftarrow \emptyset$
foreach $\mathrm{R} \in \mathcal{R}$ do
    $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \mathrm{CNF}(\mathrm{R})$
foreach $\mathrm{F} \in \mathcal{F}$ do
    $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \mathrm{CC}(\mathrm{F})$
foreach $(s, p) \in \mathcal{S} \wedge p \neq 0$ do
    if $p = \infty$ then
        $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \{s\}$
    else
        $\mathrm{Soft} \leftarrow \mathrm{Soft} \cup \{s\}$
$(\mathrm{optimum}, \mathrm{model}) \leftarrow \mathrm{PartialMinUNSAT}(\mathrm{Hard}, \mathrm{Soft})$
return $(\mathrm{optimum}, \mathrm{model})$
```

Remark: As desribed before we use a transformation like Tseitin or Plaisted-Greenbaum instead of $\mathrm{CNF}(\mathrm{R})$ in practice. Even though the Tseitin and Plaisted-Greenbaum transformations are only equi-satisfiable, this is not an issue for MaxSAT when converting formulas into hard clauses. Since the Tseitin and Plaisted-Greenbaum transformations share the same models on the original variables, one can easily verify that the search space between the converted and the original instance remains the same.

Extensions: The described use case can be extended as follows:

1. **User constraints**: A user can add additional constraints considered as hard clauses.
   If, e.g., mainboard M_1 is selected, the user definitely wants video card V_2 to be selected. But if mainboard M_2 is selected, the user definitely wants video card V_5 to be selected. Then we add the rules $(\mathrm{M}_1 \to \mathrm{V}_2) \wedge (\mathrm{M}_2 \to \mathrm{V}_5)$ as constraints to the rules $\mathcal{R}$.
2. **Focus on selection**: For each family an option "choose one of the selected" can be offered to add a constraint such that only positive selected members within a family will be considered during the re-configuration computation.
   E.g. if a user focuses on mainboards $\mathrm{M}_1, \mathrm{M}_3, \mathrm{M}_4$, a hard clause $(\mathrm{M}_1 \vee \mathrm{M}_3 \vee \mathrm{M}_4)$ will be added to the rules $\mathcal{R}$.

Example 4. *We continue our canonical Example 1: Table 5 shows multiple selections of members within families and a result model re-configuration. For all selections shown we choose priority* 1*, that means no selection in this example is an indispensable one.*

Family	Focus	Selections	Results
M	No	$(\mathrm{M}_1, 1), (\mathrm{M}_2, 1), (\neg\mathrm{M}_3, 1)$	M_4
V	Yes	$(\mathrm{V}_1, 1), (\mathrm{V}_2, 1)$	V_1
C	No	$(\mathrm{C}_2, 1), (\mathrm{C}_3, 1)$	C_4
P	No		P_1
CD	No	$(\neg\mathrm{CD}_1, \infty)$	CD_3
CR	No		CR_2

Table 5: Users selections and results

Result: *We have to make 5 changes minimally to regain a valid configuration. Without the focus set for the video cards family V, we would have to make 4 changes minimally, e.g. by choosing* M_2, V_5, C_2, P_1, CD_3, CR_2.

5.2 Use Case: Re-Configuration of the selections with priorities

In the previous use case we treated all soft clauses as equivalent. A user may prefer one member over the other, which results in priorization of the selected members. We can handle priorities with Partial Weighted MaxSAT solving. The encoding for this use case is basically the same as before, but now we bring priorities into play. Algorithm 4 shows the complete computation procedure.

Algorithm 4: Re-Configuration of a (partial) selection with priorities

```
Input: $(\mathcal{R}, \mathcal{F}, \mathcal{S})$
Output: $(\mathrm{optimum}, \mathrm{model})$, where $\mathrm{optimum}$ is the minimal
        number of priority points to change to regain a valid
        configuration and $\mathrm{model}$ is a model for the optimum
$\mathrm{Hard} \leftarrow \emptyset$
$\mathrm{Soft} \leftarrow \emptyset$
foreach $\mathrm{R} \in \mathcal{R}$ do
    $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \mathrm{CNF}(\mathrm{R})$
foreach $\mathrm{F} \in \mathcal{F}$ do
    $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \mathrm{CC}(\mathrm{F})$
foreach $(s, p) \in \mathcal{S} \wedge p \neq 0$ do
    if $p = \infty$ then
        $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \{s\}$
    else
        $\mathrm{Soft} \leftarrow \mathrm{Soft} \cup \{(s, p)\}$
$(\mathrm{optimum}, \mathrm{model}) \leftarrow$
$\mathrm{PartialWeightedMinUNSAT}(\mathrm{Hard}, \mathrm{Soft})$
return $(\mathrm{optimum}, \mathrm{model})$
```

Extension: All extensions presented in Subsection 5 carry over to this use case.

Example 5. *We reconsider our re-configuration Example 4 and add a priority of 2 for member* V_2. *Table 6 shows our selections with the corresponding weights in parentheses and the results.*

Family	Focus	Selections	Results
M	No	$(\mathrm{M}_1, 1), (\mathrm{M}_2, 1), (\neg\mathrm{M}_3, 1)$	M_4
V	Yes	$(\mathrm{V}_1, 1), (\mathrm{V}_2, 2)$	V_2
C	No	$(\mathrm{C}_2, 1), (\mathrm{C}_3, 1)$	C_4
P	No		P_1
CD	No	$(\neg\mathrm{CD}_1, \infty)$	CD_3
CR	No		CR_2

Table 6: Users selections with priorities and results

Result: *We have to change 5 priority points minimally to regain a valid configuration. If we would still be choosing member* V_1 *instead of member* V_2 *we would have to change 6 priority points, because of the higher priority of* V_2.

5.3 A MaxSAT aided re-configuration process

We reconsider the process of Figure 1 in Step 3a. UNSAT where the user gets the feedback that her current selections lead to an invalid configuration. With a SAT solver only, the user has to try by herself which selections have to be undone to regain a valid configuration. But now, we can help the user at this point by using re-configuration with MaxSAT. Figure 2 illustrates both Use Cases 5.1 and 5.2 embedded in a product configuration process using MaxSAT.

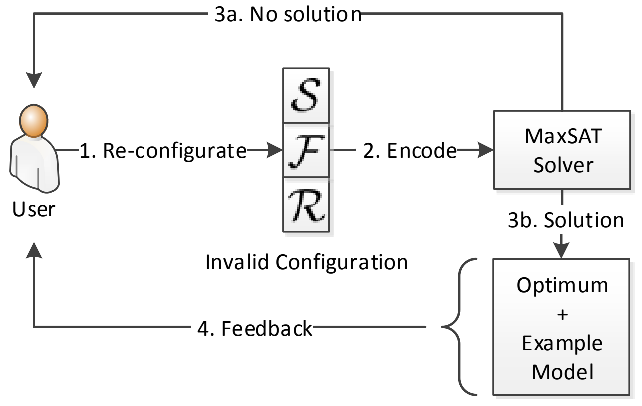


Figure 2. MaxSAT aided configuration process

After the user gets the feedback UNSAT, she can start a re-configuration of her current selections. This results in two cases:

1. **No solution:** If the indispensable selections (with priority ∞) collide with the rules or the family constraints, then there is no solution. In this case, the user has to weaken some of the indispensable selections in order to make a re-configuration possible. The user can use high priorities to weaken the desired members to ensure they will be preferred over other selections.
2. **Solution:** If the indispensable selections can be satisfied, then there exists a solution with an optimum for the prioritized selections. In this case, the user will be told about the optimum, i.e. about the number of minimal changes to regain a valid configuration. Also, an example model with the optimum will be given to the user.

Remark: Similiar to the SAT aided configuration process the following holds: If the given complete example model $l_1 \wedge \ldots \wedge l_n$ in the solution case does not satisfy the demands of the user, she can exclude this model by adding the hard clause $\neg l_1 \vee \ldots \vee \neg l_n$. Then another model with the same optimum will be produced, if one exists. If there is no other model with the same optimum, the next best optimum under the new conditions will be computed with an example model.

In case there is no solution and a user just do not want to weaken her selections with priority ∞, we can consider weakening the rules. In the next section, we will describe this possibility in detail.

5.4 Use Case: Re-configuration of rules

It is possible that the selections a user made have no solution when trying to re-configure them. Assuming the rules themselves are not contradictory, then the cause for no solution are too many selections with priority ∞. There are two cases which can occur or both at the same time:

1. **Violation of the family constraints:** If a user selects more than one member of a family with infinity priority, the family constraints are violated.
2. **Violation of rules:** If a user does not violate the family constraints, then the selected members with priority infinity are in collision with the rules.

The first case can be handled by a product configurator by simply not allowing to choose more than one member with priority infinity or giving the user a warning message when doing so.

In the second case, if the user is not willing to soften her selections, we can not re-configure the selections w.r.t. the rules. But when we have a closer look at the rules, there may be some rules, which we can soften, e.g. when a rule is not a physical or technical restriction, but only exists for marketing or similiar purposes. A company may be willing to violate or change some of these rules to build the product. Knowing the miminum number of rule changes in order to permit a desired vehicle configuration can help in managing the set of marketing rules.

For this use case, we extend Definition 1 by an additional mapping $\mathcal{S}_{\mathcal{R}} : \mathcal{R} \rightarrow (\mathbb{N}_{\geq 0} \cup \{\infty\})$, which represents the priorities of the rules a user made. After softening some of the rules this way we can re-configure the rules by maximizing the number of satisfied rules, respectively violating only a minimal number of rules. Algorithm 5 shows this procedure more formally.

Algorithm 5: Re-Configuration of rules
```
Input: $(\mathcal{R}, \mathcal{F}, \mathcal{S})$
Output: $(\mathrm{optimum}, \mathrm{model})$, where optimum is the minimal
        number of changes to regain a valid configuration and
        model is a model for the optimum
$\mathrm{Hard} \leftarrow \mathcal{S}$
$\mathrm{Soft} \leftarrow \mathcal{S}$
foreach $\mathrm{F} \in \mathcal{F}$ do
    $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \mathrm{CC(F)}$
foreach $\mathrm{R} \in \mathcal{R} \wedge \mathcal{S}_{\mathcal{R}}(\mathrm{R}) \neq 0$ do
    if $p = \infty$ then
        $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \mathrm{CNF(R)}$
    else
        $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \mathrm{CNF}(b_R \rightarrow \mathrm{R})$
        $\mathrm{Soft} \leftarrow \mathrm{Soft} \cup \{b_R\}$
foreach $(s, p) \in \mathcal{S} \wedge p \neq 0$ do
    if $p = \infty$ then
        $\mathrm{Hard} \leftarrow \mathrm{Hard} \cup \{s\}$
    else
        $\mathrm{Soft} \leftarrow \mathrm{Soft} \cup \{s\}$
$(\mathrm{optimum}, \mathrm{model}) \leftarrow \mathrm{PartialMinUNSAT}(\mathrm{Hard}, \mathrm{Soft})$
return $(\mathrm{optimum}, \mathrm{model})$
```

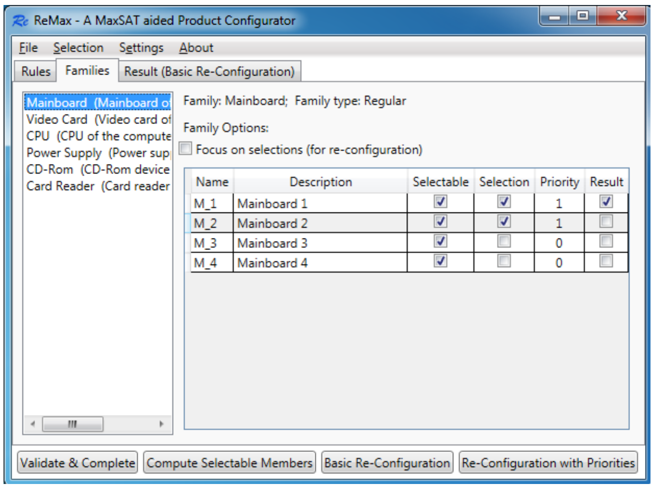
Since a rule R is an arbitrary formula, we can not just convert R to its CNF and add the resulting clauses as soft clauses. In general, some of these clauses will be satisfied and some not. Instead we want to maximize the number of rules. In other words, we are facing a *group MaxSAT problem* [2, 6], where each $\mathrm{CNF(R)}$ is a group of clauses. The goal of group MaxSAT is to satisfy the maximum number of groups. A group is satisfied if all clauses within the group are satisfied.

The group MaxSAT problem can be reduced to a partial MaxSAT problem as follows: For each non-indispensable rule R we introduce a *new* variable b_R and add the hard clauses $\mathrm{CNF}(b_R \rightarrow \mathrm{R})$. Additionally we add a unit soft clause $\{b_R\}$ for each new variable. Each satisfied variable b_R implies the whole group of clauses in $\mathrm{CNF}(\mathrm{R})$ to be satisfied. Therefore, satisfying a maximal number of the newly introduced variables satisfies a maximal number of the corresponding formulas. On the other hand, with the help of the newly introduced variables, we can identify, from the resulting model, which formulas are satisfied and show this result to the user. For a more detailed explanation, see [2, 6].

Extension: Of course, rules can also have different priorities and we can extend this use case by assigning priorities to rules and selections to compute the maximal sum of priority points. This extension can be realized analogously as described for Use Case 5.2, thus we will not describe it explicitly.

6 Implementation techniques

We implemented the above SAT-based and MaxSAT-based use cases in one product configurator — called *ReMax* — on top of our uniform logic framework, which we use for commercial applications within the context of automotive configuration. Our SAT solver provides an incremental and decremental interface. We maintain two versions (Java and .NET) and decided to implement ReMax using .NET 4.0 with C# along with the WPF Framework for the GUI. We implemented state-of-the-art partial (weighted) MaxSAT solvers Fu&Malik, PM2 and WPM1 on top of our SAT solver [5, 1].



Family: Mainboard; Family type: Regular

Family Options:

☐ Focus on selections (for re-configuration)

Name	Description	Selectable	Selection	Priority	Result
M_1	Mainboard 1	✓	✓	1	✓
M_2	Mainboard 2	✓	✓	1	☐
M_3	Mainboard 3	✓	☐	0	☐
M_4	Mainboard 4	✓	☐	0	☐

Figure 3. Screenshot of ReMax with open "Families" tab

Figure 3 shows an example screenshot from the ReMax GUI with the "Families" tab opened.

7 Experimental Results

Table 7 show statistics about the real configuration data from two different German car manufacturers, called `M01` and `M02`, which we used for our benchmarks. Car manufacturer `M01` uses arbitrary formulas as rules, whereas `M02` uses clauses as rules.

	Rules		Families	
Problem	Quantity	#Variables	Quantity	Avg. size
`M01_01`	2074	1772	34	34,294
`M01_02`	2430	2087	41	39,293
`M01_03`	1137	880	30	18,233
`M02_01`	11627	996	188	6,282
`M02_02`	4465	612	174	5,321

Table 7: Statistics about car manufacturer problems

For the following benchmarks we used two partial weighted MaxSAT solvers, which are based on the following principles:

1. **WPM1:** An unsat core-guided approach with iterative SAT calls. In each iteration a new blocking variable will be added to each soft clause within the unsat core [1].
2. **msu4:** An unsat core-guided approach with iterative SAT calls using a reduced number of blocking variables [11].

We implemented WPM1 on top of our own SAT solver while msu4 is an external solver[3].

Our environment for the benchmarks has the following hardware and software settings. Processor: Intel Core i7-3520M, 2,90 GHz; Main memory: 8 GB. WPM1, based on .NET 4.0, runs under Windows 7 while msu4 runs under Ubuntu 12.04.

For Use Case 5.2 we created three categories as follows: Out of 30%, 50% and 70% of the families one member is selected randomly with a random priority between 1 and 10. The rules have infinity priority. In general, this leads to an invalid configuration because the rules are violated. For each category we created 10 instances.

Table 8 shows the results for each category as average time in seconds. The abbreviation "exc." means that the time limit of 30 minutes was exceeded. As we can see, msu4 performs very well in all categories with reasonable times from less than one second up to about 25 seconds. Our solver WPM1 also has reasonable times from about 2 seconds up to about 28 seconds, but exeeds the time limit in two categories for the instance `M02_01`.

	30%		50%		70%	
Problem	WPM1	msu4	WPM1	msu4	WPM1	msu4
`M01_01`	7,34	0,66	12,70	1,08	15,59	1,84
`M01_02`	8,59	0,74	16,48	1,32	27,44	2,96
`M01_03`	2,10	0,33	4,10	0,45	5,80	0,85
`M02_01`	20,99	2,16	exc.	5,91	exc.	24,45
`M02_02`	3,90	0,48	9,60	1,56	13,01	4,77

Table 8: Results of Use Case 5.2 scenario

For Use Case 5.4 we created three categories as follows: Out of 30%, 50% and 70% of the families one member is selected randomly with infinity priority, which leads to an invalid configuration in general because the rules are violated. But this time, we assign all rules a priority of 1. For each category we created 10 instances.

Table 9 shows the results for each category as average time in seconds. As we can see, both solvers can handle all instances in each category in reasonable time. While WPM1 takes from about 3 seconds up to about 72 seconds, the external solver msu4 takes from less than one second up to about 9 seconds in the worst case.

[3] `http://logos.ucd.ie/web/doku.php?id=msuncore`

	30%		50%		70%	
Problem	WPM1	msu4	WPM1	msu4	WPM1	msu4
M01_01	9,35	2,39	16,19	3,93	20,63	4,35
M01_02	12,86	2,80	19,32	5,47	27,82	4,82
M01_03	2,54	0,78	5,71	1,45	6,76	1,74
M02_01	18,40	4,43	41,16	8,33	71,29	8,55
M02_02	5,13	0,49	9,88	1,04	16,32	1,48

Table 9: Results of Use Case 5.4 scenario

8 Related Work

Another approach for re-configuration uses answer set programming (ASP) on a decidable fragment of first-order logic [4]. Hence the used language is more expressive. With the growing performance of SAT solvers in the last decade, SAT solving in turn has been used to solve problem instances of ASP [8].

An algorithm for computing minimal *diagnoses* using a conflict detection algorithm is introduced in [13]. A minimal subset Δ of constraints is called a diagnosis if the original constraints without Δ are consistent. Although this approach is described for constraints of first-order sentences, the technqiues can be generalized to a wide range of other logics.

The indicated idea above is further improved in [3], where an algorithm — called FastDiag — is introduced which computes a preferred minimal diagnosis while improving performance.

We did not consider works dealing with explanations like MUS (Minimal Unsatisifable Subset) iteration. When using MUS iteration for re-configuration, a user not only has to manually solve each conflict, but also will not necessarily solve the conflicts in an optimal manner, i.e. only changing a minimal number of selections.

9 Conclusion

We described product configuration for propositional logic based rule sets which are widely used in the automotive industry. We showed applications of SAT solving by two use cases. Furthermore, we showed use cases of how MaxSAT can be used for product configuration when it comes to an invalid configuration. With MaxSAT we are able to re-configure an invalid configuration in an optimal way, i.e. we can compute the minimal number of necessary changes. We embedded both scenarios in configuration processes showing how a user can be guided during the configuration process.

We presented an implementation of a product configurator — ReMax — supporting all of the described use cases using state-of-the-art SAT and MaxSAT solving techniques. From real automotive configuration data from two different German premium car manufacturers we created synthetic product configuration benchmarks for the presented use cases. Besides our own MaxSAT solver we used the external solver msu4 to measure and compare the performance. As our experimental results show, we can re-configure those problem instances in reasonable time. Since some problem instances could be solved within a few seconds, our product configurator could be used as an interactive tool in these cases. Other problem instances took over a minute in the worst case, but is still more than adequate for a responsive batch service. While this may seem long, we were told that the manual configuration of an order without tool support by a trial and error process may well take on the order of half an hour.

We do not claim that our approach is currently fit for use as a consumer configurator. However, many business units of a car manufacturer, such as engineering or after sales are in need of a (re-)configurator that feeds directly off the engineering product documentation. E.g., many test prototypes must be built before start of production with a varying set of options.

Expert users sometimes need some complete car configurations which cover all valid combinations of a subset of options, e.g. for testing purposes. With a SAT based (re-)configurator, an expert user can start the configuration from the desired options instead of tediously following the given configuration process in a usual sales configurator. At any time, the user can ask the configurator for "any completion" or, using MaxSAT, for a "minimal completion" of the partial configuration to a complete configuration.

REFERENCES

[1] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy, 'Solving (weighted) partial MaxSAT through satisfiability testing', in *Theory and Applications of Satisfiability Testing - SAT 2009*, ed., Oliver Kullmann, volume 5584 of *Lecture Notes in Computer Science*, 427–440, Springer Berlin Heidelberg, (2009).

[2] Josep Argelich and Felip Many, 'Exact Max-SAT solvers for over-constrained problems.', *Journal of Heuristics*, **12**(4–5), 375–392, (September 2006).

[3] A. Felfernig, M. Schubert, and C. Zehentner, 'An efficient diagnosis algorithm for inconsistent constraint sets', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **26**(1), 53 – 62, (2012).

[4] Gerhard Friedrich, Anna Ryabokon, Andreas A. Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner, '(re)configuration using answer set programming', in *IJCAI-11 Configuration Workshop Proceedings*, eds., Kostyantyn Shchekotykhin, Dietmar Jannach, and Markus Zanker, pp. 17–24, Barcelona, Spain, (July 2011).

[5] Zhaohui Fu and Sharad Malik, 'On solving the partial MAX-SAT problem', in *Theory and Applications of Satisfiability Testing—SAT 2006*, eds., Armin Biere and Carla P. Gomes, volume 4121 of *Lecture Notes in Computer Science*, 252–265, Springer Berlin Heidelberg, (2006).

[6] Federico Heras, Antnio Morgado, and Joo Marques-Silva, 'An empirical study of encodings for group MaxSAT', in *Canadian Conference on AI*, eds., Leila Kosseim and Diana Inkpen, volume 7310 of *Lecture Notes in Computer Science*, pp. 85–96. Springer, (2012).

[7] Wolfgang Küchlin and Carsten Sinz, 'Proving consistency assertions for automotive product data management', *Journal of Automated Reasoning*, **24**(1–2), 145–163, (2000).

[8] Fangzhen Lin and Yuting Zhao, 'ASSAT: Computing answer sets of a logic program by SAT solvers.', *Artifical Intelligence*, **157**(1–2), 115–137, (August 2004).

[9] Peter Manhart, 'Reconfiguration – a problem in search of solutions', in *IJCAI-05 Configuration Workshop Proceedings*, eds., Dietmar Jannach and Alexander Felfernig, pp. 64–67, Edinburgh, Scotland, (July 2005).

[10] João Marques-Silva, 'Practical applications of boolean satisfiability', in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, 74–80, IEEE, (2008).

[11] João Marques-Silva and Jordi Planes, 'Algorithms for maximum satisfiability using unsatisfiable cores', in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pp. 408–413. IEEE, (2008).

[12] David A. Plaisted and Steven Greenbaum, 'A structure-preserving clause form translation', *Journal of Symbolic Computation*, **2**(3), 293–304, (September 1986).

[13] Raymond Reiter, 'A theory of diagnosis from first principles', *Artificial Intelligence*, **32**(1), 57 – 95, (April 1987).

[14] Carsten Sinz, 'Towards an optimal CNF encoding of boolean cardinality constraints', in *Principles and Practice of Constraint Programming—CP 2005*, ed., Peter van Beek, Lecture Notes in Computer Science, 827–831, Springer Berlin Heidelberg, (2005).

[15] G. S. Tseitin, 'On the complexity of derivations in the propositional calculus', *Studies in Constructive Mathematics and Mathematical Logic*, **Part II**, 115–125, (1968).

[16] Rouven Walter, Christoph Zengler, and Wolfgang Küchlin, 'Applications of MaxSAT in automotive configuration', in *Proceedings of the 15th International Configuration Workshop*, eds., Michel Aldanondo and Andreas Falkner, pp. 21–28, Vienna, Austria, (August 2013).