algorithms

Article

Joining Constraint Satisfaction Problems and Configurable CAD Product Models: A Step-by-Step Implementation Guide

Paul Christoph Gembarski

Institute of Product Development, Leibniz University Hanover, An der Universität 1, 30823 Garbsen, Germany; gembarski@ipeg.uni-hannover.de

Abstract: In configuration design, the task is to compose a system out of a set of predefined, modu-lar building blocks assembled by defined interfaces. Product configuration systems, both with or without integration of geometric models, implement reasoning techniques to model and explore the resulting solution spaces. Among others, the formulation of constraint satisfaction problems (CSP) is state of the art and the informational background in many proprietary configuration engine software packages. Basically, configuration design tasks can also be implemented in modern computer aided design (CAD) systems as these contain different techniques for knowledge-based product modeling but literature reports only little about detailed application examples, best practices or training materials. This article aims at bridging this gap and presents a step-by-step implementation guide for CSP-based CAD configurators for combinatorial designs with the example of Autodesk Inventor.

Keywords: knowledge-based engineering; product configurators; constraint satisfaction problems; computer aided design; solution space modeling

check for updates

Citation: Gembarski, P.C. Joining Constraint Satisfaction Problems and Configurable CAD Product Models: A Step-by-Step Implementation Guide. *Algorithms* **2022**, *15*, 318. https://doi.org/10.3390/a15090318

Academic Editor: Natanael Karjanto

Received: 8 July 2022
Accepted: 5 September 2022
Published: 6 September 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The design of variant-rich, configurable products has not just been a topic since the rise of Mass Customization during the 1990s [1,2]. In configuration design, the task is to compose a system out of a set of predefined, modular building blocks assembled by defined interfaces [3,4]. Depending on count and quality of the interfaces, a small number of building blocks already leads to huge solution spaces, i.e., the set of all feasible configurations. Taking the example of Durhuus and Eilers [5], six LEGO bricks in the size 2 to 4 span a solution space of 915,103,765 possible configurations when the only restriction is that all blocks need somehow to be connected to at least another one in the usual rectangular grid. Assuming that an average designer would need around 5 min to assemble the six bricks in a computer aided design (CAD) system, derive a technical drawing with bill-of-materials and check-in the data into a product data management system, the manual documentation of all configurations would take approx. 8705 years, working 24/7. In automotive engineering, the manifold of solution spaces is even magnitudes larger [6].

Automating such (routine) design tasks is one aim of knowledge-based engineering, with product configurators as an instance for its application [7,8]. One of the first documented product configuration systems dates back to the 1980s, which is McDermott's R1/XCON rule-based configurer [9]. While this system in its final version used thousands of production rules as inference engine to get from requirements to the needed product variant, other approaches rely on model-based reasoning techniques, like the formulation of constraint satisfaction problems (CSP) [10,11]. In brief, a CSP consists of a set of finite domains as containers for variables and their possible parameter values as well as a set of constraints that are links between the domains and determine which combinations of values of the single variables are allowed [12]. Compared to rule-based systems, the CSP approach usually leads to better performance, reduced maintenance effort and higher implementation efficiency [13].

There exists a multitude of proprietary software packages to integrate constraint-based configuration and CAD [14,15]. However, many of today's CAD systems offer plenty of functionalities already in their out of the box version to implement knowledge-based designs and configurators [16]. These functionalities include, among others, design rules, mathematical and logical constraints between parameters and design features, parameter control via spreadsheets and script languages, as well as of course the application programming interface (API) [17–19]. A CAD-based design automation system using these functionalities can perform the above task of assembling the six LEGO bricks in less than two days on today's desktop computers. Nonetheless, the implementation of CSP-based configuration designs needs a level of abstraction since the initial design problem needs to be transformed into a configuration problem [20]. Afterwards, the information system needs to be designed accordingly to this.

Engineering literature reports only little about detailed application examples, best practices or training materials related to this topic. Existing workbooks usually are related to expert system shells from the 1990s, e.g., PLAKON [21,22] or the above-mentioned proprietary software packages. In order to foster and promote the possibilities and application of constraint-based configuration and to encourage engineers in using this approach for configurable assemblies, this contribution presents a step-by-step implementation guide for CSPs into combinatorial CAD models. The CAD assembly is set up in the CAD system Autodesk Inventor, the constraint solver is implemented through its API.

The article is structured as follows: In Section 2, the theoretical background of constraint satisfaction problems as a model-based reasoning technique is briefly illustrated, before Section 3 introduces the running example for the configuration design task. Section 4 then presents a specification of the information system to be built. In Section 5, the code for the CSP implementation is developed, including an introduction of new domains and constraints afterwards. The coupling to the CAD model is discussed in Section 6. The final Section 7 concludes the article.

2. Theoretical Background

Basically, inference mechanisms as part of a knowledge-based engineering system can be implemented on the basis of model elements and their relations. Depending on how both are modeled, literature distinguishes logic-based, constraint-based and resource-based approaches [13].

For a logic-based representation of knowledge and reasoning, an object-oriented data structure is used [23]. This consists of the actual objects (so-called individuals), logical structures for aggregating objects (concepts) and binary relationships between individual objects (roles). Objects, in a narrow sense, may be understood as classes, from which arbitrarily many instances can be derived and assigned different values. Additionally, the class concept allows attributes of objects to be inherited from higher hierarchy levels. The reasoning is performed by checking whether a special object is included in a superordinate class [11].

Constraint-based approaches represent, as mentioned in the introduction, constraint networks where a constraint represents the relation between two model elements and has a rule for value assignment [13]. In this way, e.g., physical or engineering contexts can be modeled, e.g., the calculation of a bolted connection. Values applied to the constraint network can then be propagated, which means that the values of all other model elements are calculated on their basis. By nature, it is irrelevant according to which variables the formulas are resolved and which variables are used as inputs [24]. CSPs in this application correspond to a system of equations, constraint propagation to the solution of the system of equations.

A further very typical task for constraint-based reasoning is the representation of configuration problems. For this, the elements are modeled likewise with their value ranges and relations. An example for this is the so-called map coloring problem [25]. As shown in Figure 1, the single countries are abstracted into domains containing the available colors for the map as values. Inequality constraints are introduced between two neighboring countries, expressing that neighboring countries are to be colored differently [26].

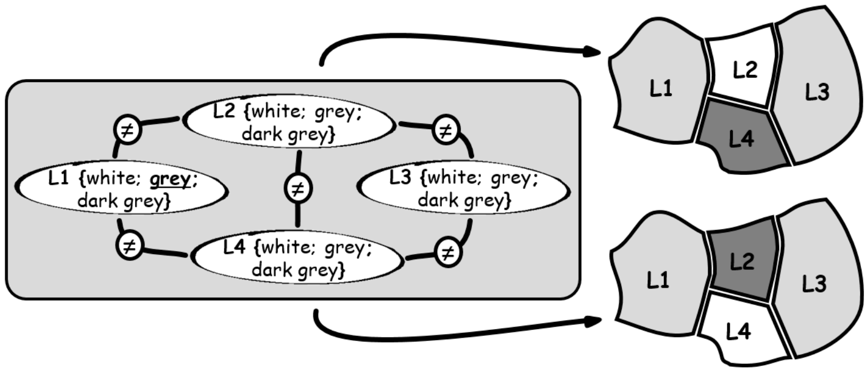

Figure 1. Map coloring problem (initial value for constraint propagation highlighted in L1).

A constraint solver now has the task of finding solutions for the constraint network that satisfy all constraints. For this purpose, different algorithms are available, e.g., Generate-and-Test, which systematically tests possible parameter combinations. However, this method also calculates predictably erroneous value assignments. In the example of the map in Figure 1, *L1* has an initial value *grey*. The solver then sequentially assigns colors to the other domains and tests the involved constraints. Starting with *L2* = *white*, the constraint between *L1* and *L2* is satisfied. Proceeding to *L3* = *white*, a constraint is violated, the solution is discarded. The algorithm will then return to the last known state that satisfied all constraints and change the last variable assignment. If *L3* is then assigned *gray* instead of *white*, the problem can be further processed and solved. This procedure is called backtracking [26].

Since a constraint network can be written as a graph (the domains form the nodes, the constraints the arcs), alternative algorithms exploit graph-theoretic methods to reduce the effective search space by enforcing local consistency. A CSP is considered node-consistent if all unary constraints, which limit the value of a single variable, are satisfied. Arc consistency involves binary constraints between domains. Inconsistent values from the further involved domain are then immediately removed for the search [12]. For the example in Figure 1, this means that the value *gray* can no longer be available for *L2* and *L4* because *L1* is already occupied by *gray* as input. Path consistency extends this by not only collapsing one domain but removing pairs of values from a constraint [26]. Continuing the above example with pre-collapsed *L2* and *L4*, *gray* remains as a single consistent value for *L3* and leads automatically to the two solutions depicted on the right of the figure, without testing further configurations.

Resource-based reasoning is a special case of constraint-based reasoning. Here, the relationships between the model elements are represented on the basis of resource consumption and provision functions. The reasoning mechanism then aims at achieving a state of equilibrium. The concept is kept abstract and counts also, e.g., installation space, assembly time or technical interfaces to resources [27].

For use in a design automation system or in constraint-based configuration, such reasoning mechanisms need to be coupled with a product model [20]. Such a product model can be a realized as a simple catalog from which options are chosen by the algorithm. Another implementation aims at the generation of a geometric (CAD) model for visualization or further processing in the sense of design templates [28,29]. Note that model-based reasoning in general is able to include configurable artifacts beside the actual product model. On the one hand, this may be, e.g., the joint configuration of product features and their respecting manufacturing processes [30,31], but also combines product and service configuration in hybrid offerings [32].

3. Modeling Task

The running example for this article is a configurable toy sorting box for small kids as depicted in Figure 2. The customer first chooses four different slots that are described by *article number*, *shape* and *color*. In a second configuration step, the customer can choose a brick variant for the selected slots, which is described by *article number*, *color*, *shape* and *infill* (flat, raised, embossed—refer, e.g., to the blue prism bricks in Figure 2).

Figure 2. Configurable toy sorting box with according brick variants.

For the slots, ten different variants are available. The main restriction is that no equal slots may be chosen. Permutations (*red*, *yellow*, *green*, *blue* OR *red*, *blue*, *green*, *yellow*) are to be allowed. Nonetheless, if the customer by incident chooses a predefined template configuration (three out of four connected colors), the last color should automatically be restricted to the missing template slot. Depending on the chosen slot, there may be up to three possible brick variants for the second configuration step.

4. Specification of the Information System

The resulting customer interaction during configuration is depicted in the use case diagram in Figure 3. Besides choosing slots and bricks, the customer should furthermore be able to change or reset a choice or the whole configuration as such at each point of the configuration process. In order to keep the configuration process flexible, it should not matter in which sequence the slots are selected by the user.

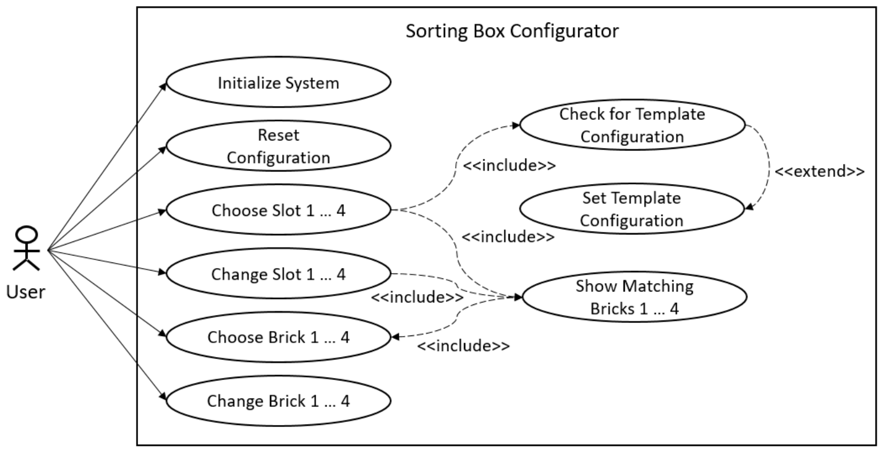

Figure 3. Use case diagram for customer interaction.

The abstraction of the design problem to a configuration problem results in a first set of four domains *SLOT1* . . . *SLOT4* representing the slots with their respective alternatives. A second set of four domains *BRICK1* . . . *BRICK4* contains the representation of all brick variants.

As constraints, *inequality* of *SLOT1*, *SLOT2*, *SLOT3* and *SLOT4* to each other is expressed by bidirectional constraints. The bidirectionality leads to the fact that it later does not matter which value must be set first and which must be propagated afterwards. Additionally, *equalities* between the slots and the corresponding bricks are introduced. A special feature of the configuration tool is the template definition of four predefined colors. Therefore, an auxiliary domain *TEMPLATE* is created that contains the template configurations. This constraint is only to be propagated if three slot domains, independently which of them, have an assigned value, otherwise the constraint is relaxed. The resulting constraint network is depicted in Figure 4.

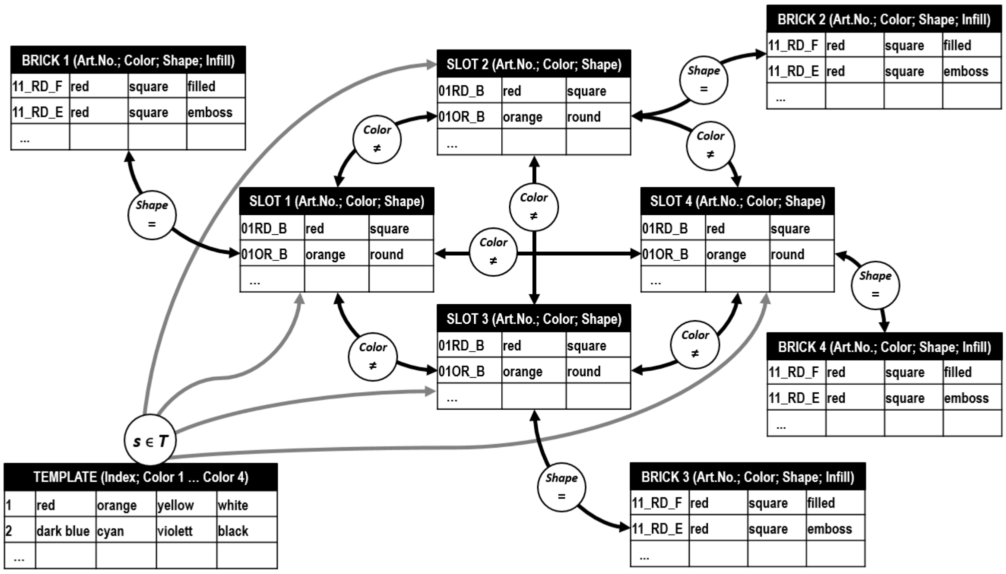

Figure 4. Constraint network for the sorting box configuration problem.

In order to avoid hardcoding of domains, variables and constraints, domain and constraint lists should be stored externally and so be easily updateable and extendable. After starting the configuration tool, the first step is that domain and constraint lists are loaded from this repository and the constraint network is established automatically in the blackboard of the configuration tool (Figure 5).

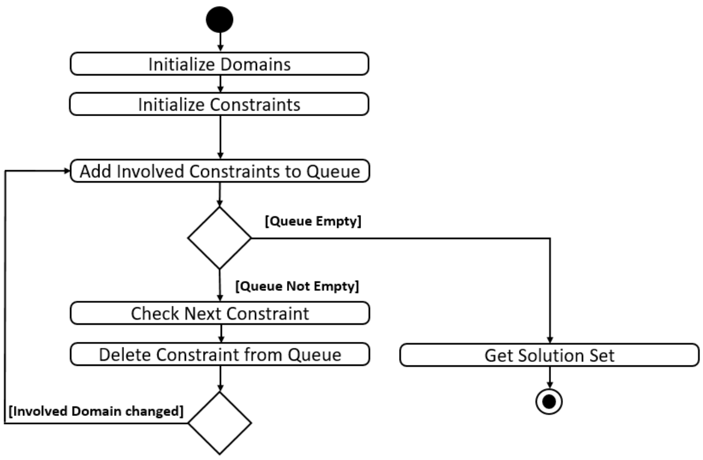

Figure 5. Activity diagram for constraint handling.

The system then waits for user inputs and adds all involved constraints linked to the changed domain to a queue. As long as the queue is not empty, the solver sequentially propagates and deletes the next constraint in the queue. Afterwards, in order to apply local consistency, the system checks whether neighboring domains have been affected. If so, again the involved constraints are added to the end of the queue. After the queue has been completely processed, the solution set is output.

5. Building the CSP

The implementation is designed for Autodesk Inventor Professional as representative for parametric CAD systems with knowledge-based product modeling and a Visual Basic for Applications (VBA) API, which is commonly available as basic API language in many CAD systems. Although this language has many deficiencies and more similarities to scripting automation, this origin in macros still keeps it widespread in the mechanical engineering domain and practically relevant. As Inventor additionally offers the possibility to integrate and embed MS Excel spreadsheets into the CAD model, Excel was chosen as the repository for domain and constraint lists. Other spreadsheet applications are not supported in this context. So, the CSP-based configurator is implemented as VBA macro with its own user interface in order to control both Excel and Inventor for the later CAD assembly.

As a domain can be abstracted as a list of variables and their values, dictionaries are a feasible representation as they offer some advantages over, e.g., array lists: First, all dictionary properties are writable and retrievable (including the key values) and can handle nearly every type of key except arrays. Second, dictionary methods offer the possibility to check if a key value already exists. Finally, although slower in initial creation, dictionaries are significantly faster in processing than array lists. In order to use dictionaries, the MS scripting runtime within VBA must be activated.

The code for the example is available in the appendix of this article and additionally via https://doi.org/10.25835/6wdfdp03 (last accessed on 4 September 2022). The link contains the Autodesk Inventor 2020 and Excel files as well as separate VBA files (form, basic and class files) with the code for the sorting box with four slots and four bricks. The data set is reduced to an empty assembly and the macro. No part files or geometry is included. So, the macro in this form should also work as it is with other CAD systems which use VBA as scripting language by importing the corresponding VBA files into the macro editor.

5.1. Domain and Constraint Lists

Beside the VBA form for the configuration system, the user interface also contains the repository for the domain and constraint lists. Regarding the two domains for slots and bricks, Figure 6 shows the structure in the Excel spreadsheet. Each domain is situated on its own workbook containing all describing properties like *Article Number*, *Color*, *Shape* and, in the case of the bricks, additionally the *Infill* as columns with headlines.

A	B	C
Article	**Color**	**Shape**
01RD_B	red	square
01OR_B	orange	round
01GN_B	green	hexagon
01DB_B	dark blue	triangle
01YW_B	yellow	octagon
01VT_B	violet	five star
01CY_B	cyan	eight star
01WH_B	white	ellipse
01BK_B	black	ring

A	B	C	D
Article	**Color**	**Shape**	**Infill**
11_RD_F	red	square	filled
11_RD_E	red	square	emboss
11_OR_F	orange	round	filled
11_OR_E	orange	round	emboss
11_GN_F	green	hexagon	filled
11_DB_F	dark blue	triangle	filled
11_DB_E	dark blue	triangle	emboss
11_YW_F	yellow	octagon	filled
11_VT_F	violet	five star	filled

Figure 6. Spreadsheet definition for slot (**left**) and brick (**right**) domain.

As can be seen from the constraint network, the constraint list has to store equality, inequality and the obligatory assignment of the fourth color from a template. This is coded as *Expression* as depicted in Figure 7. Additionally, the requirement of a flexible configuration process and local consistency leads to the distinction of both unary and binary constraints. In order to maintain a uniform declaration, a single constraint can include up to four arguments. Two are needed for equality and inequality, i.e., the two domains that are linked by the constraint. Four (the colors of the template) are needed for handling the obligatory assignment.

	A	B	C	D	E	F	G
1	**Key**	**Expression**	**BiDirectional**	**Domain_A**	**Domain_B**	**Domain_C**	**Domain_D**
2	1	unequal	TRUE	Slots1	Slots2		
3	2	unequal	TRUE	Slots1	Slots3		
4	3	unequal	TRUE	Slots1	Slots4		
5	4	unequal	TRUE	Slots2	Slots3		
6	5	unequal	TRUE	Slots2	Slots4		
7	6	unequal	TRUE	Slots3	Slots4		
8	7	obligation	FALSE	red	orange	yellow	violet
9	8	obligation	FALSE	dark blue	violet	white	cyan
10	9	equal	TRUE	Slots1	Bricks1		
11	10	equal	TRUE	Slots2	Bricks2		
12	11	equal	TRUE	Slots3	Bricks3		
13	12	equal	TRUE	Slots4	Bricks4		

Figure 7. Spreadsheet constraint definition.

5.2. Creating Domains and the Constraint List

Before the actual handling of the input data can take place, the initial variable declarations need to be done in VBA (Appendix A, row 1 to 21). Beside the dictionaries for the eight domains, two additional dictionaries *dict_SlotsORI* (for original) and *dict_BricksORI* are declared. These domain masters are intended as reference objects to be written in the blackboard of the configuration system. E.g., if a domain needs to be reset when the user changes his or her selection, the domain master is copied in the respecting domain and all constraints are to be regenerated without reobtaining data from the spreadsheet.

Furthermore, the dictionaries *dict_ConList* and *dict_ConQue* are declared for constraint handling. The first is meant as a reference list containing all constraints, the second is the actual queue. Finally, a global counter is declared, which is needed for the template constraint and global strings as the internal working memory for assigned domain values.

Additionally, as part of the variable declaration, three public class modules *iComponent_Bricks*, *iComponent_Slots* and *iConstraint* are introduced as follows in Figure 8. The first two define the domains and their properties, which are translated to the columns of the dictionaries later. The last class module is the generalized constraint variable. The dictionaries *ConArg1* ... *ConArg4* represent domains that are connected to a constraint, the strings formalize value assignments. This is, e.g., necessary for the template configurations where these arguments contain the four colors. Additionally, the string *ConExpr* contains the type of the constraint, *ConDir*, whether it is unary or binary.

The subroutine for reading the inventory and creating the blackboard is shown in Appendix A, row 62 to 76. Beside the path to the spreadsheet, it basically contains three functions that address the single workbooks. Note that the path to the repository in row 62 needs to be adjusted to the path where the repository is locally saved.

The generation of the domains can be found in Appendix A, row 79 to 109 for the slots and row 112 to 143 for the bricks. In the first part of both routines, the domain master is created. Basically, it adds each row from row two on (first row is the column headline) of the spreadsheet to the dictionary. Thereby, *ArticleID* is a temporary variable which contains the key for the dictionary. Reading the spreadsheet ends when an empty cell in the first column of the spreadsheet is reached. After the reference is created, it is distributed to the four respecting domains as the initial solution set.

Populating the constraint reference list works similar, as shown in Appendix A, row 145 to 271. The key is realized as a counter. In order to couple domains with equality and inequality constraints (Appendix A, row 154 to 249), the involved domains need to be declared as arguments. As VBA does not allow to compose variable names out of, e.g., a combination of strings and numeric values at runtime, the strings obtained from the spreadsheet once need to be related to the dictionaries. Therefore, the select case expressions contain the corresponding assignments for the constraint arguments. Additionally to the dictionaries, the arguments are also stored as strings for further processing. If the constraint is binary, a second constraint with opposite arguments is created.

```
Option Explicit

'Order Number
Public Bricks_ArticleID As String

'Properties
Public Bricks_Colour As String
Public Bricks_Shape As String
Public Bricks_Infill As String
```

```
Option Explicit

'Order Number
Public Slots_ArticleID As String

'Properties
Public Slots_Colour As String
Public Slots_Shape As String
```

```
Option Explicit

'Domain Arguments
Public ConArg1 As Dictionary
Public ConArg1Str As String
Public ConArg2 As Dictionary
Public ConArg2Str As String
Public ConArg3 As Dictionary
Public ConArg3Str As String
Public ConArg4 As Dictionary
Public ConArg4Str As String
'Type
Public ConExpr As String
'Binary?
Public ConDir As String
'Index from Spreadsheet
Public ConIndex As Integer
```

Figure 8. Declaration of public classes.

If the spreadsheet contains a constraint with the expression *Obligation*, the four arguments of the corresponding entry in the constraint list are populated with the four colors as shown in Appendix A, row 251 to 271.

5.3. User Interface

Following the use case diagram from Figure 3, the user form needs to contain control elements for system initialization (command button *Generate Domains* which starts *Sub read_inventory*), resetting the whole configuration process (command button *Reset All* which clears all domains and outputs), as well as choosing and resetting of slots and bricks and generating the CAD model (Figure 9).

Primary user interaction elements are list boxes. After initialization, the configuration system writes the domain content as items into the respective list box. The user then can double click on an item to make his or her selection which is written into a label (e.g., *red*, *yellow* and *green* in Figure 9) and the remaining list boxes are updated after constraint handling. For each selection, a command button *relax* is implemented which resets just the according domain and updates the others after constraint handling. The VBA code for the user form including the naming of control elements is documented in Appendix B.

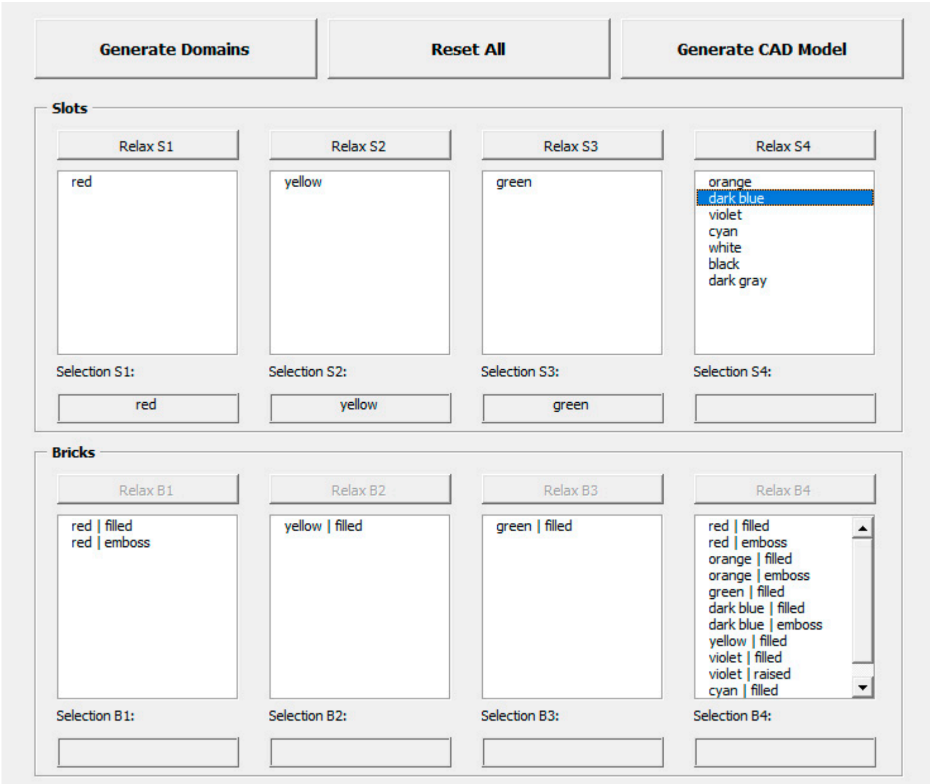

Figure 9. User form for sorting box configuration.

5.4. Constraint Handling

5.4.1. Queueing

After initialization, the queue is at first empty. If now the user chooses a slot by double clicking on an entry in a list box, the code in Appendix B, row 45 to 69, first establishes node consistency as it processes the unary constraint for collapsing the respecting domain *SLOT1*. Afterwards, it increases the counter for the selected domains by one which is relevant for the queueing of the obligatory assignments.

Afterwards, the actual queueing takes place as the configuration system scans through the constraint reference list from the blackboard and adds either all equality/inequality constraints to the queue where the involved domain is the first argument (source) or it adds the obligation constraints if three of four slots have been selected by the user. The code for the other list box inputs is similar.

5.4.2. Solver and Output of the Solution Set

The constraint solver itself is mainly organized within a while loop that is processed until the queue is empty (Appendix A, row 323 to 369). Therefore, the first constraint from the queue is taken and read out in a temporary variable set. Afterwards, the constraint is deleted from the queue and, depending on the expression of the constraint, the arguments are used to call the according functions.

The function *Del_Equal_Slots* (Appendix A, row 484 to 498) eliminates values that are equal to those of a source domain and is used to process the inequality constraints. The function *Bricks_to_Slots* (Appendix A, row 501 to 516) processes the equality constraints as it eliminates all values that are unequal to those of a source domain (Figure 10).

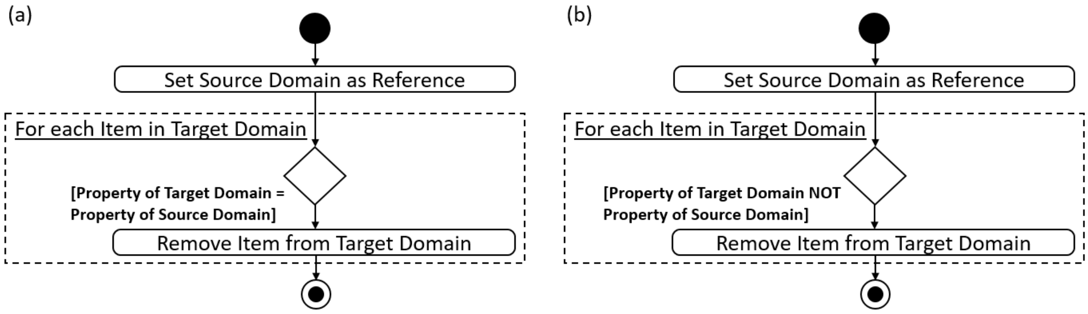

Figure 10. Activity diagram for inequality (**a**) and equality (**b**) constraint handling.

Handling the obligatory assignments is a bit more complicated since it should not matter which order or domain assignment the three chosen are. Figure 11 depicts the structure of the corresponding code (Appendix A, row 518 to 625).

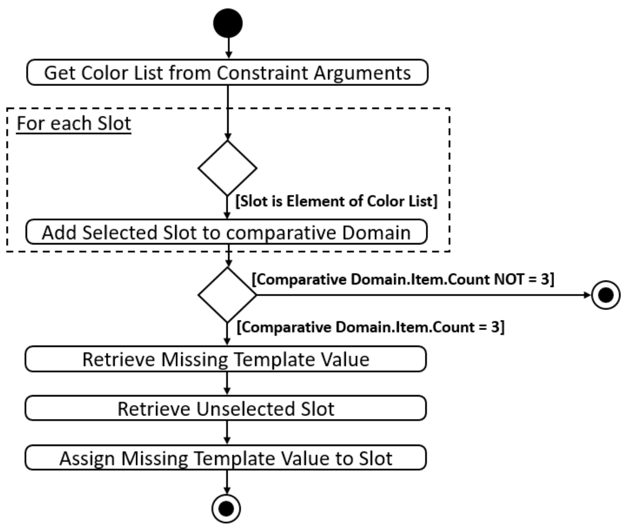

Figure 11. Activity diagram for handling the obligatory assignment of fourth template slot.

The final output of the solution set is then organized as an update of the list boxes after the queue has been processed (Appendix A, row 274 to 320).

5.5. Resetting a Selection

In order to reset an already made choice, Figure 12 describes the activities of the configuration system. At first, the domain to be reset is cleared and repopulated from the domain master. If the reset domain is of type *slots*, the corresponding *brick* domain is reset as well. In the next step, all domains that have already been processed by the user and that have a selection are reevaluated by propagating all unary constraints assigned to it. In this way, the according solution set is extended by the selection which has been reset. Afterwards, each constraint from the constraint list is evaluated if the actual domain is the source, i.e., first argument. If so, the constraint is added to the queue. After all domains have been reevaluated, the constraint solver is activated and the reset is completed. The corresponding code is shown in Appendix A, row 372 to 468.

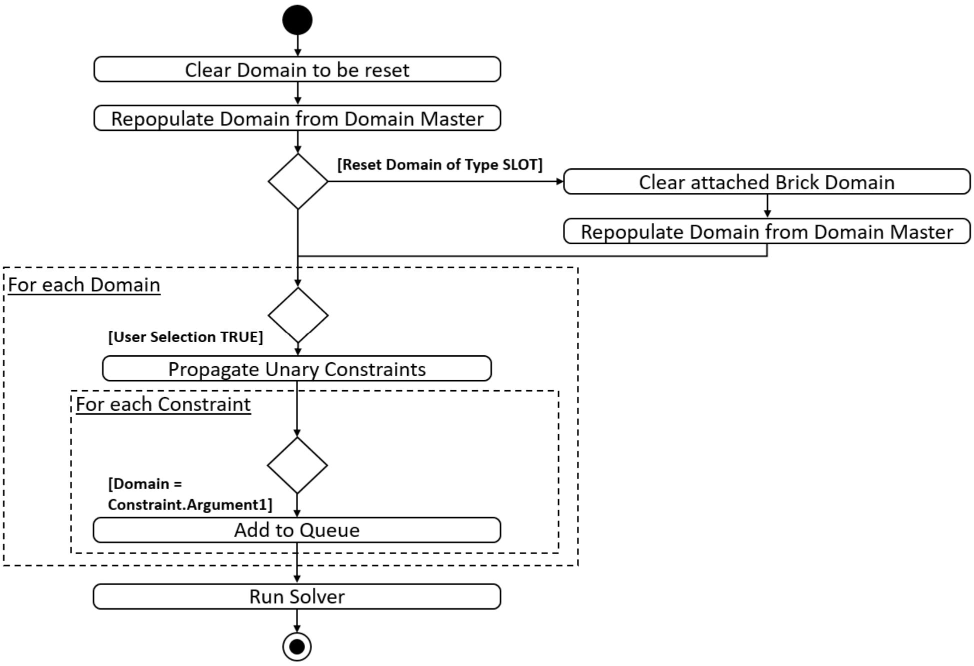

Figure 12. Activity diagram for relaxing constraints and update.

5.6. Extending the Example from 8 to 10 Domains

In order to show a typical maintenance task beside the adaption of the domain content, the following steps describe the extension of the existing CSP with each a fifth slot and brick domain.

At first, the user interface in *userform1* is extended by the relevant control elements which are copied and renamed according to the existing naming convention (e.g., into *lbx_slots5*).

Regarding the code in the module *main* (Appendix A), the first step is to add the corresponding new variable declarations for *dict_Slots5* and *dict_Bricks5* according to row 5 and 8. Other extensions involve:

- *Sub reset*: New domains and control elements integrated according to Appendix A, row 26, 31, 40, 44, 48 and 52.
- *Sub generate domains*: New loop according to Appendix A, row 96 to 98 and row 130 to 132.
- *Sub get_constraints*: New translations like in Appendix A, row 160/161, 168/169, 180/181, 188/189, 208/209, 214/215, 227/228, 235/236.
- *Sub update_listboxes*: Introduction of *slots5* and *bricks5* according to Appendix A, row 277–282 and 300–304.
- *Sub constraint_relax*: Introduction of *slots5* and *bricks5* according to Appendix A, row 375–387 and 426–434.

Afterwards, the new control elements need to be equipped with their event handlers:

- *cmd_relax_1_5*: Code added according to Appendix B, row 8–16.
- *lbx_slots5_doubleclick*: Code added according to Appendix B, row 45–69.

At last, the constraint list in the spreadsheet needs to be extended with four inequality constraints linking each *SLOT1*, *SLOT2*, *SLOT3* and *SLOT4* to *SLOT5* as well as an equality constraint linking *SLOT5* and *BRICKS5*.

The above changes do not include the extension of the obligatory assignment. Here, the spreadsheet and the constraint handling need to be added by a fifth domain argument DOMAIN_E.

6. Addressing the CAD Assembly

Basically, two different options for the integration of the CAD model are available. On the one hand, following the principle of the *digital master*, an assembly document contains all possible part and feature occurrences that are either activated or suppressed according to the configured product. On the other hand, a *model* or *draft generator* starts with an empty assembly document and adds all parts and features from the configured product to it. An advantage of the former is that the assembly is created completely in advance, i.e., including all positioning dependencies or geometric constraints between components. Accordingly, the advantage of the latter is that there is no need for an assembly to be predefined and engineered in advance, but the positioning of each part needs to be automated instead. In the case of the sorting box, the configuration system merges both principles since there are no duplicate selections.

When the user approves his or her choice and initializes the generation of the CAD model, the configuration system opens the digital master for the sorting box assembly, as shown in Figure 13. Afterwards, any activated parts are suppressed to assure an explicit initial state. The system then processes each slot and transmits the according user selection to the CAD model where the corresponding slot part is activated. Afterwards, this part is positioned in the fixed grid according to its position (*SLOT1* is upper left, *SLOT2* upper right, etc.). This avoids including redundant slot parts at each position. After the slots have been activated and positioned, the configuration system processes the bricks accordingly (Figure 14).

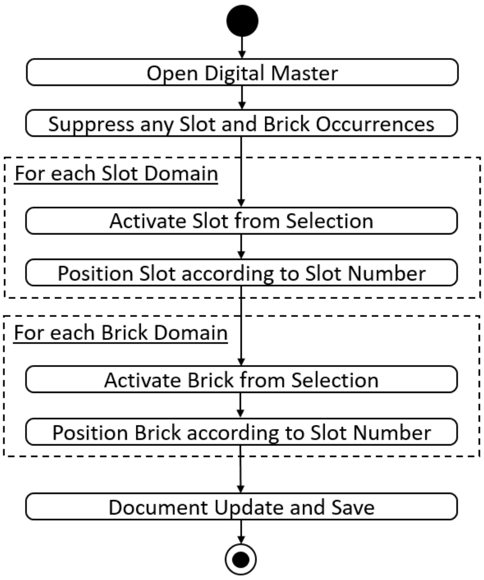

Figure 13. Activity diagram for CAD model update.

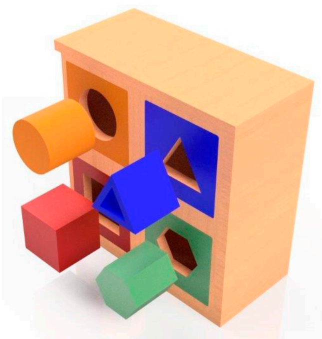

Figure 14. Configured sorting box.

7. Discussion and Conclusions

Focusing on the above configuration problem, the core of the example, the configuration of the four slots mirrors exactly the map coloring problem which has been extended by additional related choices and the obligatory assignment of an option depending on selections made by the user. Thus, the example of the sorting box is a placeholder for many configurable assemblies and products. Taking online car configurators, the functionalities of mandatory choices, configuration interdictions and the realization of templates or lines of equipment are similar functions. The search strategy for the solution is kept simple in this example by enforcing local consistency, without further examining the time complexity of the solution algorithm. User input or a modification of a domain leads to adding the corresponding constraints to the queue without further nesting or querying. In this case, there is no need for backtracking. Further constraints could be hierarchical constraints or additional classifying criteria, e.g., if two light colors are chosen, only two dark colors may be added, which correspond widely to the obligatory assignments.

Due to the separation of knowledge base, i.e., the spreadsheet, inference engine, i.e., the API code, and the CAD model, a more or less linear control flow is realized. A function which has not been included is math constraints since these can also be expressed directly in the geometric CAD model by, e.g., relating parameters by calculations. In this context, a more complex control flow might integrate reobtaining parameter values from the CAD model, e.g., values of dimensions, and feed them back into the CSP. An open question is if modeling complex assemblies like known from plant engineering, which contain several hundred domains, is adequate with this kind of implementation and the search strategy or if the performance will suffer compared to configurators based on proprietary software packages.

A deficiency of the presented example is the fact that VBA does not allow generating variables or code at runtime. Other languages, e.g., python or expert system shells, which include this functionality, will shorten the code and raise efficiency and maintainability. Furthermore, especially code generation and an automatic adaptation of the user interface at runtime could widely automate user interface creation, e.g., to extend the sorting box to five slots even with less effort. In the same way, e.g., python offers a constraint module which includes a backtracking solver, recursive backtracking solver and minimum conflicts solver as well as 10 different constraint types. A module for remote controlling Autodesk Inventor is also available, a comparative study of implementing the above design task is an obvious avenue.

The example of the sorting box can easily be extended by, e.g., pricing information that is calculated with respect to the user selections. Therefore, the domains need to be extended by the corresponding part costs, which have to be added by the configuration system. As an example, to include non-geometric features like services, it would be conceivable to choose between a neutral standard package and a package that shows the actual configured sorting box. The described extension to 10 domains involves multiple adjustments to the code. From a theoretical perspective, an extension to 1000 domains seems not adequate. However, taking into account that many combinatorial design tasks in mechanical engineering can be broken down to 10 to 25 core components on an assembly level, a relevant field of application exists, independently from the implementation in VBA or a higher language.

Further interesting issues lie in coupling different inference mechanisms to the CSP-based configuration system and decentralizing its knowledge base as the CAD system itself can control, e.g., part or feature occurrences by rules or other scripting. Additionally, case-based reasoning would be an interesting approach to include a dynamic generation of template configurations depending on, e.g., configuration history, user ratings or overall most favorite configurations. Including data about the user, this could be further developed to a recommendation system.

Concluding, the application of CSP-based configuration using just the functionalities of a standard CAD system has the potential of automating combinatorial designs and integrate knowledge-based CAD assemblies with high efficiency. The approach is compatible with existing and usually familiar tools, thus easy in its application and extendible. In such a way, managing the combinatorial complexity of a design solution space is possible.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The code for the example, the Autodesk Inventor assembly file and the excel repository is publicly available via https://doi.org/10.25835/6wdfdp03 (last accessed on 4 September 2022).

Conflicts of Interest: The author declares no conflict of interest.

Appendix A Code for Module *Main*

Table A1. VBA Code for Module *Main*.

```
'For Dictionary Use activate:
'Extras -> References -> Microsoft Scripting Runtime

    'Generate Dictionaries for all Components
    Public dict_Slots1 As New Dictionary, dict_Slots2 As New Dictionary
    Public dict_Slots3 As New Dictionary, dict_Slots4 As New Dictionary
    Public dict_SlotsORI As New Dictionary
    Public dict_Bricks1 As New Dictionary, dict_Bricks2 As New Dictionary
    Public dict_Bricks3 As New Dictionary, dict_Bricks4 As New Dictionary
    Public dict_BricksORI As New Dictionary

    'Generate Constraint Handling
    Public dict_ConList As New Dictionary
    Public dict_ConQue As New Dictionary

    'Internal Selection Working Memory
    Public strSlot1, strSlot2, strSlot3, strSlot4 As String
    Public strBrick1, strBrick2, strBrick3, strBrick4 As String

    'Global Counter
    Public count_Slots As Integer


Sub reset()
    'Clear all Dicitonaries
    dict_Slots1.RemoveAll
    dict_Slots2.RemoveAll
    dict_Slots3.RemoveAll
    dict_Slots4.RemoveAll
    dict_SlotsORI.RemoveAll
    dict_Bricks1.RemoveAll
    dict_Bricks2.RemoveAll
    dict_Bricks3.RemoveAll
    dict_Bricks4.RemoveAll
    dict_BricksORI.RemoveAll
    dict_ConQue.RemoveAll
    dict_ConList.RemoveAll

    'Clear User Interface
    UserForm1.lbx_Slots1.Clear
    UserForm1.lbx_Slots2.Clear
```

Table A1. *Cont.*

Row	Code
42	UserForm1.lbx_Slots3.Clear
43	UserForm1.lbx_Slots4.Clear
44	UserForm1.lbx_Bricks1.Clear
45	UserForm1.lbx_Bricks2.Clear
46	UserForm1.lbx_Bricks3.Clear
47	UserForm1.lbx_Bricks4.Clear
48	UserForm1.lbl_Slots1.Caption = “”
49	UserForm1.lbl_Slots2.Caption = “”
50	UserForm1.lbl_Slots3.Caption = “”
51	UserForm1.lbl_Slots4.Caption = “”
52	UserForm1.lbl_Bricks1.Caption = “”
53	UserForm1.lbl_Bricks2.Caption = “”
54	UserForm1.lbl_Bricks3.Caption = “”
55	UserForm1.lbl_Bricks4.Caption = “”
56	
57	‘Reset Global Counter
58	count_Slots = 0
59	End Sub
60	
61	
62	Sub read_inventory()
63	Dim ExcWB As Excel.Workbook
64	‘————————————————————-
65	‘ADJUST PATH TO LOCAL PATH OF THE REPOSITORY FILE BELOW!
66	‘————————————————————-
67	Set ExcWB = Workbooks.Open(“ . . . \Inventory_List.xlsx”)
68	
69	‘Call creating subs
70	generate_domains ExcWB.Worksheets(“Slots”)
71	generate_Brickss ExcWB.Worksheets(“Bricks”)
72	get_constraints ExcWB.Worksheets(“_CONSTRAINTS_”)
73	
74	‘Close Workbook
75	ExcWB.Close
76	End Sub
77	
78	
79	Sub generate_domains(ByVal ExcWS As Excel.WorkSheet)
80	Dim iRow As Integer
81	Dim key As Variant
82	
83	‘Set Start for reading in second row
84	iRow = 2
85	‘Get Slot Domain Master
86	Do Until ExcWS.Cells(iRow, 1).Value = “”
87	ArticleID = ExcWS.Cells(iRow, 1).Value
88	Set oPart1 = New iComponent_Slots
89	oPart1.Slots_Colour = ExcWS.Cells(iRow, 2).Value
90	oPart1.Slots_Shape = ExcWS.Cells(iRow, 3).Value
91	dict_SlotsORI.Add ArticleID, oPart1
92	iRow = iRow + 1
93	Loop
94	
95	‘Distribute Master to all 4 Slot Domains
96	For Each key In dict_SlotsORI
97	dict_Slots1.Add (key), dict_SlotsORI(key)
98	Next
99	For Each key In dict_SlotsORI

Table A1. *Cont.*

```
        dict_Slots2.Add (key), dict_SlotsORI(key)
    Next
    For Each key In dict_SlotsORI
        dict_Slots3.Add (key), dict_SlotsORI(key)
    Next
    For Each key In dict_SlotsORI
        dict_Slots4.Add (key), dict_SlotsORI(key)
    Next
End Sub
Sub generate_Brickss(ByVal ExcWS As Excel.WorkSheet)
    Dim iRow As Integer
    Dim key As Variant
    ‘Set Start for reading in second row
    iRow = 2
    ‘Get Brick Domain Master
    Do Until ExcWS.Cells(iRow, 1).Value = ““
       ArticleID = ExcWS.Cells(iRow, 1).Value
       Set oPart1 = New iComponent_Bricks
       oPart1.Bricks_Colour = ExcWS.Cells(iRow, 2).Value
       oPart1.Bricks_Shape = ExcWS.Cells(iRow, 3).Value
       oPart1.Bricks_Infill = ExcWS.Cells(iRow, 4).Value
       dict_BricksORI.Add ArticleID, oPart1
       iRow = iRow + 1
    Loop
    ‘Distribute Master to all 4 Brick Domains
    For Each key In dict_BricksORI
        dict_Bricks1.Add (key), dict_BricksORI(key)
    Next
    For Each key In dict_BricksORI
        dict_Bricks2.Add (key), dict_BricksORI(key)
    Next
    For Each key In dict_BricksORI
        dict_Bricks3.Add (key), dict_BricksORI(key)
    Next
    For Each key In dict_BricksORI
        dict_Bricks4.Add (key), dict_BricksORI(key)
    Next
End Sub
Sub get_constraints(ByVal ExcWS As Excel.WorkSheet)
    Dim ConPa As iConstraint
    ‘Set Start for reading in second row
    iRow = 2
    KeyID = 1
    ‘Get Constraints from Repository
    Do Until ExcWS.Cells(iRow, 1).Value = ““
    ‘—— EQUALITIES / INEQUALITIES ——
    If ExcWS.Cells(iRow, 2) = “equal” Or ExcWS.Cells(iRow, 2) = “unequal” Then
        ‘Formalize for Processing in VBA
        Set oConstraint1 = New iConstraint
```

Table A1. *Cont.*

```
'Assign Source Domain
Select Case ExcWS.Cells(iRow, 4)
    Case "Slots1"
        Set oConstraint1.ConArg1 = dict_Slots1
    Case "Slots2"
        Set oConstraint1.ConArg1 = dict_Slots2
    Case "Slots3"
        Set oConstraint1.ConArg1 = dict_Slots3
    Case "Slots4"
        Set oConstraint1.ConArg1 = dict_Slots4
    Case "Bricks1"
        Set oConstraint1.ConArg1 = dict_Bricks1
    Case "Bricks2"
        Set oConstraint1.ConArg1 = dict_Bricks2
    Case "Bricks3"
        Set oConstraint1.ConArg1 = dict_Bricks3
    Case "Bricks4"
        Set oConstraint1.ConArg1 = dict_Bricks4
End Select
oConstraint1.ConArg1Str = ExcWS.Cells(iRow, 4)
'Assign Target Domain
Select Case ExcWS.Cells(iRow, 5)
    Case "Slots1"
        Set oConstraint1.ConArg2 = dict_Slots1
    Case "Slots2"
        Set oConstraint1.ConArg2 = dict_Slots2
    Case "Slots3"
        Set oConstraint1.ConArg2 = dict_Slots3
    Case "Slots4"
        Set oConstraint1.ConArg2 = dict_Slots4
    Case "Bricks1"
        Set oConstraint1.ConArg2 = dict_Bricks1
    Case "Bricks2"
        Set oConstraint1.ConArg2 = dict_Bricks2
    Case "Bricks3"
        Set oConstraint1.ConArg2 = dict_Bricks3
    Case "Bricks4"
        Set oConstraint1.ConArg2 = dict_Bricks4
End Select
oConstraint1.ConArg2Str = ExcWS.Cells(iRow, 5)
'Get Expression
oConstraint1.ConExpr = ExcWS.Cells(iRow, 2)
'Add to Constraint List
dict_ConList.Add KeyID, oConstraint1
KeyID = KeyID + 1

'If binary get inverse Constraint
If ExcWS.Cells(iRow, 3) = "TRUE" Then
Set oConstraint1 = New iConstraint
Select Case ExcWS.Cells(iRow, 5)
    Case "Slots1"
        Set oConstraint1.ConArg1 = dict_Slots1
    Case "Slots2"
        Set oConstraint1.ConArg1 = dict_Slots2
    Case "Slots3"
        Set oConstraint1.ConArg1 = dict_Slots3
    Case "Slots4"
        Set oConstraint1.ConArg1 = dict_Slots4
```

Table A1. *Cont.*

Code

```
            Case “Bricks1”
                Set oConstraint1.ConArg1 = dict_Bricks1
            Case “Bricks2”
                Set oConstraint1.ConArg1 = dict_Bricks2
            Case “Bricks3”
                Set oConstraint1.ConArg1 = dict_Bricks3
            Case “Bricks4”
                Set oConstraint1.ConArg1 = dict_Bricks4
        End Select
        oConstraint1.ConArg1Str = ExcWS.Cells(iRow, 5)
        Select Case ExcWS.Cells(iRow, 4)
            Case “Slots1”
                Set oConstraint1.ConArg2 = dict_Slots1
            Case “Slots2”
                Set oConstraint1.ConArg2 = dict_Slots2
            Case “Slots3”
                Set oConstraint1.ConArg2 = dict_Slots3
            Case “Slots4”
                Set oConstraint1.ConArg2 = dict_Slots4
            Case “Bricks1”
                Set oConstraint1.ConArg2 = dict_Bricks1
            Case “Bricks2”
                Set oConstraint1.ConArg2 = dict_Bricks2
            Case “Bricks3”
                Set oConstraint1.ConArg2 = dict_Bricks3
            Case “Bricks4”
                Set oConstraint1.ConArg2 = dict_Bricks4
        End Select
        oConstraint1.ConArg2Str = ExcWS.Cells(iRow, 4)
        oConstraint1.ConExpr = ExcWS.Cells(iRow, 2)
        dict_ConList.Add KeyID, oConstraint1
        KeyID = KeyID + 1
        End If
    End If

    ‘—— OBLIGATIONS ——
    If ExcWS.Cells(iRow, 2) = “obligation” Then
        Set oConstraint1 = New iConstraint
        ‘Get Arguments
        oConstraint1.ConArg1Str = ExcWS.Cells(iRow, 4)
        oConstraint1.ConArg2Str = ExcWS.Cells(iRow, 5)
        oConstraint1.ConArg3Str = ExcWS.Cells(iRow, 6)
        oConstraint1.ConArg4Str = ExcWS.Cells(iRow, 7)
        ‘Get Get Expression
        oConstraint1.ConExpr = ExcWS.Cells(iRow, 2)
        oConstraint1.ConIndex = ExcWS.Cells(iRow, 1)
        ‘Add to Constraint List
        dict_ConList.Add KeyID, oConstraint1
        KeyID = KeyID + 1
    End If

    iRow = iRow + 1
    Loop
    count_Slots = 0

End Sub
```

Table A1. *Cont.*

```
Sub update_Listboxes()
    'Update Content of the User Interface
    'Update Slots
    UserForm1.lbx_Slots1.Clear
    'Repopulate Listbox from Domain
    For Each key In dict_Slots1
        Set oComp = dict_Slots1(key)
        UserForm1.lbx_Slots1.AddItem (oComp.Slots_Colour)
    Next
    UserForm1.lbx_Slots2.Clear
    For Each key In dict_Slots2
        Set oComp = dict_Slots2(key)
        UserForm1.lbx_Slots2.AddItem (oComp.Slots_Colour)
    Next
    UserForm1.lbx_Slots3.Clear
    For Each key In dict_Slots3
        Set oComp = dict_Slots3(key)
        UserForm1.lbx_Slots3.AddItem (oComp.Slots_Colour)
    Next
    UserForm1.lbx_Slots4.Clear
    For Each key In dict_Slots4
       Set oComp = dict_Slots4(key)
        UserForm1.lbx_Slots4.AddItem (oComp.Slots_Colour)
    Next

    'Update Bricks
    UserForm1.lbx_Bricks1.Clear
    For Each key In dict_Bricks1
        Set oComp = dict_Bricks1(key)
        UserForm1.lbx_Bricks1.AddItem (oComp.Bricks_Colour & " | " & oComp.Bricks_Infill)
    Next
    UserForm1.lbx_Bricks2.Clear
    For Each key In dict_Bricks2
        Set oComp = dict_Bricks2(key)
        UserForm1.lbx_Bricks2.AddItem (oComp.Bricks_Colour & " | " & oComp.Bricks_Infill)
    Next
    UserForm1.lbx_Bricks3.Clear
    For Each key In dict_Bricks3
        Set oComp = dict_Bricks3(key)
        UserForm1.lbx_Bricks3.AddItem (oComp.Bricks_Colour & " | " & oComp.Bricks_Infill)
    Next
    UserForm1.lbx_Bricks4.Clear
    For Each key In dict_Bricks4
        Set oComp = dict_Bricks4(key)
        UserForm1.lbx_Bricks4.AddItem (oComp.Bricks_Colour & " | " & oComp.Bricks_Infill)
    Next
End Sub


Public Sub Constraint_Solver()

    Dim key As Variant
    Dim arg1 As Dictionary, arg2 As Dictionary, arg3 As Dictionary, arg4 As Dictionary
    Dim arg1str, arg2str, arg3str, arg4str As String
    Dim ConExpr As String
    Dim oComp As iComponent_Slots
    Dim ConPa As iConstraint
    'As long as Queue is not empty
```

Table A1. *Cont.*

```
    Do While dict_ConQue.Count <> 0
        For Each key In dict_ConQue
            ‘Get Arguments from Constraint in Queue
            Set ConPa = dict_ConQue(key)
            Set arg1 = ConPa.ConArg1
            Set arg2 = ConPa.ConArg2
            Set arg3 = ConPa.ConArg3
            Set arg4 = ConPa.ConArg4
            arg1str = ConPa.ConArg1Str
            arg2str = ConPa.ConArg2Str
            arg3str = ConPa.ConArg3Str
            arg4str = ConPa.ConArg4Str
            ConExpr = ConPa.ConExpr
            ‘Get Expression for processing
            Select Case ConExpr
                ‘Unequality
                Case “unequal”
                    ‘Delete Constraint from Queue
                    dict_ConQue.Remove (key)
                    ‘Call Processing
                    Call Del_Equal_Slots(arg1, arg2)
                ‘Equality
                Case “equal”
                    dict_ConQue.Remove (key)
                    Call Bricks_to_Slots(arg1, arg2)
                ‘Obligatory Assignment
                Case “obligation”
                    dict_ConQue.Remove (key)
                    Call Set_Obliged_Slots(arg1str, arg2str, arg3str, arg4str)
                Case Else
                    dict_ConQue.Remove (key)
            End Select
        Next
    Loop
    ‘Update User Interface
    update_Listboxes

End Sub


Sub Constraint_Relax()
‘—— Repopulte reset Domain ——
    ‘Evaluate if Domain is NOT collapsed
    If strSlot1 = ““ Then
        ‘Refresh the domain including reset selection
        ‘for slot
        dict_Slots1.RemoveAll
        For Each key In dict_SlotsORI
            dict_Slots1.Add (key), dict_SlotsORI(key)
        Next
        ‘for brick
        dict_Bricks1.RemoveAll
        For Each key In dict_BricksORI
            dict_Bricks1.Add (key), dict_BricksORI(key)
        Next
    End If

```

Table A1. *Cont.*

Code

```
If strSlot2 = "" Then
    dict_Slots2.RemoveAll
    For Each key In dict_SlotsORI
        dict_Slots2.Add (key), dict_SlotsORI(key)
    Next
    dict_Bricks2.RemoveAll
    For Each key In dict_BricksORI
        dict_Bricks2.Add (key), dict_BricksORI(key)
    Next
End If

If strSlot3 = "" Then
    dict_Slots3.RemoveAll
    For Each key In dict_SlotsORI
        dict_Slots3.Add (key), dict_SlotsORI(key)
    Next
    dict_Bricks3.RemoveAll
    For Each key In dict_BricksORI
        dict_Bricks3.Add (key), dict_BricksORI(key)
    Next
End If

If strSlot4 = "" Then
    dict_Slots4.RemoveAll
    For Each key In dict_SlotsORI
        dict_Slots4.Add (key), dict_SlotsORI(key)
    Next
    dict_Bricks4.RemoveAll
    For Each key In dict_BricksORI
        dict_Bricks4.Add (key), dict_BricksORI(key)
    Next
End If

'—— Add relevant Constraints to queue ——
Dim oConstr As iConstraint
'if the domain is already occupied by a selection
'then add relevant constraints to queue where this is source
If Not strSlot1 = "" Then
    Call Un_Constraint(dict_Slots1, strSlot1)
    For Each key In dict_ConList
        Set oConstr = dict_ConList(key)
        If oConstr.ConArg1Str = "Slots1" Then
            dict_ConQue.Add (key), dict_ConList(key)
        End If
    Next
End If

If Not strSlot2 = "" Then
    Call Un_Constraint(dict_Slots2, strSlot2)
    For Each key In dict_ConList
        Set oConstr = dict_ConList(key)
        If oConstr.ConArg1Str = "Slots2" Then
            dict_ConQue.Add (key), dict_ConList(key)
        End If
    Next
End If

If Not strSlot3 = "" Then
```

Table A1. *Cont.*

```
        Call Un_Constraint(dict_Slots3, strSlot3)
        For Each key In dict_ConList
            Set oConstr = dict_ConList(key)
            If oConstr.ConArg1Str = “Slots3” Then
                dict_ConQue.Add (key), dict_ConList(key)
            End If
        Next
    End If

    If Not strSlot4 = “” Then
        Call Un_Constraint(dict_Slots4, strSlot4)
        For Each key In dict_ConList
            Set oConstr = dict_ConList(key)
            If oConstr.ConArg1Str = “Slots4” Then
                dict_ConQue.Add (key), dict_ConList(key)
            End If
        Next
    End If

    ‘Start Solver
    Constraint_Solver
End Sub


Sub Un_Constraint(ByRef arg1 As Dictionary, ByVal ConVal As String)
‘Collapses Domain according to node consistency
    Dim key As Variant
    Dim oComp As iComponent_Slots
    For Each key In arg1
        Set oComp = arg1(key)
        If Not oComp.Slots_Colour = ConVal Then
            arg1.Remove (key)
        End If
    Next
End Sub


Sub Del_Equal_Slots(ByRef arg1 As Dictionary, ByRef arg2 As Dictionary)
‘Elimitates values equal to the reference domain
    Dim key1 As Variant, key2 As Variant
    Dim oComp1 As iComponent_Slots, oComp2 As iComponent_Slots
    For Each key1 In arg1
        Set oComp1 = arg1(key1)
        For Each key2 In arg2
            Set oComp2 = arg2(key2)
            If oComp1.Slots_Colour = oComp2.Slots_Colour Then
                arg2.Remove (key2)
            End If
        Next
    Next

End Sub


Sub Bricks_to_Slots(ByRef arg1 As Dictionary, ByRef arg2 As Dictionary)
‘Elimitates values unequal to the reference domain
    Dim key1 As Variant, key2 As Variant
```

Table A1. *Cont.*

Code

```
    Dim oComp1 As iComponent_Slots
    Dim oComp2 As iComponent_Bricks
    For Each key1 In arg1
        Set oComp1 = arg1(key1)
        For Each key2 In arg2
            Set oComp2 = arg2(key2)
            If Not oComp1.Slots_Colour = oComp2.Bricks_Colour Then
                arg2.Remove (key2)
            End If
        Next
    Next

End Sub

Sub Set_Obliged_Slots(ByVal arg1str As String, ByVal arg2str As String, _
            ByVal arg3str As String, ByVal arg4str As String)

'Assigns missing value from template

    Dim key1, key2, key3, key4 As Variant
    Dim KeyA, KeyB As Variant
    Dim oComp1 As iComponent_Slots, oComp2 As iComponent_Slots, oComp3 As _
        iComponent_Slots, oComp4 As iComponent_Slots, oCompA As iComponent_Slots

    Dim dict_comp As New Dictionary
    Dim dict_set As New Dictionary

    'establish comparative dictionary from constraint arguments
    'if this domain has an assigned value write it to the comparative dictionary
    If dict_Slots1.Count = 1 Then
        For Each key1 In dict_Slots1
            Set oComp1 = dict_Slots1(key1)
            'get colour independently from sequence or position
            If oComp1.Slots_Colour = arg1str Or oComp1.Slots_Colour = arg2str Or _
                oComp1.Slots_Colour = arg3str Or oComp1.Slots_Colour = arg4str Then
                dict_comp.Add (key1), dict_Slots1(key1)
            End If
        Next
    End If
    If dict_Slots2.Count = 1 Then
        For Each key2 In dict_Slots2
            Set oComp2 = dict_Slots2(key2)
            If oComp2.Slots_Colour = arg1str Or oComp2.Slots_Colour = arg2str Or _
                oComp2.Slots_Colour = arg3str Or oComp2.Slots_Colour = arg4str Then
                dict_comp.Add (key2), dict_Slots2(key2)
            End If
        Next
    End If
    If dict_Slots3.Count = 1 Then
        For Each key3 In dict_Slots3
            Set oComp3 = dict_Slots3(key3)
            If oComp3.Slots_Colour = arg1str Or oComp3.Slots_Colour = arg2str Or _
                oComp3.Slots_Colour = arg3str Or oComp3.Slots_Colour = arg4str Then
                dict_comp.Add (key3), dict_Slots3(key3)
            End If
        Next
    End If
```

Table A1. *Cont.*

Row	Code
561	If dict_Slots4.Count = 1 Then
562	For Each key4 In dict_Slots4
563	Set oComp4 = dict_Slots4(key4)
564	If oComp4.Slots_Colour = arg1str Or oComp4.Slots_Colour = arg2str Or _
565	oComp4.Slots_Colour = arg3str Or oComp4.Slots_Colour = arg4str Then
566	dict_comp.Add (key4), dict_Slots4(key4)
567	End If
568	Next
569	End If
570	
571	‘check if three slots have been assigned
572	If dict_comp.Count = 3 Then
573	‘establish control dictionary for value retrieval
574	For Each key In dict_SlotsORI
575	Set oCompA = dict_SlotsORI(key)
576	If oCompA.Slots_Colour = arg1str Or oCompA.Slots_Colour = arg2str Or _
577	oCompA.Slots_Colour = arg3str Or oCompA.Slots_Colour = arg4str Then
578	dict_set.Add (key), dict_SlotsORI(key)
579	End If
580	Next
581	
582	‘identify missing assignment
583	For Each KeyA In dict_comp
584	dict_set.Remove (KeyA)
585	Next
586	
587	‘set assignment
588	For Each KeyA In dict_set
589	Set oCompA = dict_set(KeyA)
590	‘if slot one is the open one then assign last value here and collapse
591	If Not dict_Slots1.Count = 1 Then
592	For Each key1 In dict_Slots1
593	Set oComp1 = dict_Slots1(key1)
594	If Not oComp1.Slots_Colour = oCompA.Slots_Colour Then
595	dict_Slots1.Remove (key1)
596	End If
597	Next
598	End If
599	If Not dict_Slots2.Count = 1 Then
600	For Each key2 In dict_Slots2
601	Set oComp2 = dict_Slots2(key2)
602	If Not oComp2.Slots_Colour = oCompA.Slots_Colour Then
603	dict_Slots2.Remove (key2)
604	End If
605	Next
606	End If
607	If Not dict_Slots3.Count = 1 Then
608	For Each key3 In dict_Slots3
609	Set oComp3 = dict_Slots3(key3)
610	If Not oComp3.Slots_Colour = oCompA.Slots_Colour Then
611	dict_Slots3.Remove (key3)
612	End If
613	Next
614	End If
615	If Not dict_Slots4.Count = 1 Then
616	For Each key4 In dict_Slots4

Table A1. *Cont.*

```
                    Set oComp4 = dict_Slots4(key4)
                    If Not oComp4.Slots_Colour = oCompA.Slots_Colour Then
                        dict_Slots4.Remove (key4)
                    End If
                Next
            End If
        Next
    End If
End Sub
```

Appendix B Code for *Userform1*

Table A2. VBA Code for *Userform1*.

```
Private Sub cmd_gen_domains_Click()
'Generates the Domains from the Excel Repository and inits User Interface
    Main.reset
    Main.read_inventory
    Main.update_Listboxes
End Sub

Private Sub cmd_relax1_1_Click()
    'Reset Label in User Interface and Internal Selection Working Memory
    lbl_Slots1.Caption = ""
    strSlot1 = ""
    'decrease global counter by one
    count_Slots = count_Slots − 1
    'Refreshopen Domains
    Main.Constraint_Relax
End Sub

Private Sub cmd_relax1_2_Click()
    lbl_Slots2.Caption = ""
    strSlot2 = ""
    count_Slots = count_Slots − 1
    Main.Constraint_Relax
End Sub

Private Sub cmd_relax1_3_Click()
    lbl_Slots3.Caption = ""
    strSlot3 = ""
    count_Slots = count_Slots − 1
    Main.Constraint_Relax
End Sub

Private Sub cmd_relax1_4_Click()
    lbl_Slots4.Caption = ""
    strSlot4 = ""
    count_Slots = count_Slots − 1
    Main.Constraint_Relax
End Sub

Private Sub cmd_reset_Click()
```

Table A2. *Cont.*

```
'Empty Domains, Queue and User Interface
    reset
End Sub


Private Sub lbx_Slots1_DblClick(ByVal Cancel As MSForms.ReturnBoolean)
'Selection of a Domain Variable and Queueing of Neighbours
    'Add Selection to Label in User Interface and Internal Selection Working Memory
    lbl_Slots1.Caption = lbx_Slots1.Text
    strSlot1 = lbx_Slots1.Text
    'Add unary Constraint to Queue for Collapsing
    Call Main.Un_Constraint(dict_Slots1, strSlot1)
    'Increase global counter by one
    count_Slots = count_Slots + 1
    'Add relevant binary Constraints to Queue
    Dim oConstr As iConstraint
    For Each key In dict_ConList
        Set oConstr = dict_ConList(key)
        If oConstr.ConArg1Str = "Slots1" Then
            dict_ConQue.Add (key), dict_ConList(key)
        End If
        If count_Slots = 3 Then
            If oConstr.ConExpr = "obligation" Then
                dict_ConQue.Add (key), dict_ConList(key)
            End If
        End If
    Next
    'Start Solver
    Main.Constraint_Solver
End Sub


Private Sub lbx_Slots2_DblClick(ByVal Cancel As MSForms.ReturnBoolean)
    lbl_Slots2.Caption = lbx_Slots2.Text
    strSlot2 = lbx_Slots2.Text
    Call Main.Un_Constraint(dict_Slots2, strSlot2)
    count_Slots = count_Slots + 1
    Dim oConstr As iConstraint
    For Each key In dict_ConList
        Set oConstr = dict_ConList(key)
        If oConstr.ConArg1Str = "Slots2" Then
            dict_ConQue.Add (key), dict_ConList(key)
        End If
        If count_Slots = 3 Then
            If oConstr.ConExpr = "obligation" Then
                dict_ConQue.Add (key), dict_ConList(key)
            End If
        End If
    Next
    Main.Constraint_Solver
End Sub

Private Sub lbx_Slots3_DblClick(ByVal Cancel As MSForms.ReturnBoolean)
    lbl_Slots3.Caption = lbx_Slots3.Text
    strSlot3 = lbx_Slots3.Text
    Call Main.Un_Constraint(dict_Slots3, strSlot3)
```

Table A2. *Cont.*

Row	Code
96	count_Slots = count_Slots + 1
97	Dim oConstr As iConstraint
98	For Each key In dict_ConList
99	Set oConstr = dict_ConList(key)
100	If oConstr.ConArg1Str = "Slots3" Then
101	dict_ConQue.Add (key), dict_ConList(key)
102	End If
103	If count_Slots = 3 Then
104	If oConstr.ConExpr = "obligation" Then
105	dict_ConQue.Add (key), dict_ConList(key)
106	End If
107	End If
108	Next
109	Main.Constraint_Solver
110	End Sub
111	
112	Private Sub lbx_Slots4_DblClick(ByVal Cancel As MSForms.ReturnBoolean)
113	lbl_Slots4.Caption = lbx_Slots4.Text
114	strSlot4 = lbx_Slots4.Text

References

1. Hotz, L.; Felfernig, A.; Günter, A.; Tiihonen, J. A short history of configuration technologies. In *Knowledge-Based Configuration—From Research to Business Cases*; Felfernig, A., Hotz, L., Bagley, C., Tiihonen, J., Eds.; Morgan Kaufmann: Waltham, MA, USA, 2014; pp. 9–19.
2. Schwede, L.N.; Greve, E.; Krause, E.; Otto, K.; Moon, S.; Albers, A.; Kirchner, E.; Lachmayer, R.; Bursac, N.; Inkermann, D.; et al. How to Use the Levers of Modularity Properly—Linking Modularization to Economic Targets. *J. Mech. Des.* **2022**, *144*, 071401. [CrossRef]
3. Schreiber, G. Knowledge engineering. In *Foundations of Artificial Intelligence*; Van Harmelen, F., Lifschitz, V., Porter, B., Eds.; Elsevier: Amsterdam, The Netherlands, 2008; Volume 3, pp. 929–946. [CrossRef]
4. Gembarski, P.C. Three ways of integrating computer-aided design and knowledge-based engineering. In *Proceedings of the Design Society: DESIGN Conference*; Cambridge University Press: Cambridge, UK, 2020; Volume 1, pp. 1255–1264. [CrossRef]
5. Durhuus, B.; Eilers, S. On the entropy of LEGO®. *J. Appl. Math. Comput.* **2014**, *45*, 433–448. [CrossRef]
6. Pil, F.K.; Holweg, M. Linking product variety to order-fulfillment strategies. *Interfaces* **2004**, *34*, 394–403. [CrossRef]
7. Verhagen, W.; Bermell-Garcia, P.; van Dijk, R.; Curran, R. A critical review of Knowledge-Based Engineering: An identification of research challenges. *Adv. Eng. Inf.* **2012**, *26*, 5–15. [CrossRef]
8. Hopgood, A.A. *Intelligent Systems for Engineers and Scientists*; CRC Press: Boca Raton, FL, USA, 2016.
9. McDermott, J. R1: A rule-based configurer of computer systems. *Artif. Intell.* **1982**, *19*, 39–88. [CrossRef]
10. Aldanondo, M.; Hadj-Hamou, K.; Moynard, G.; Lamothe, J. Mass customization and configuration: Requirement analysis and constraint based modeling propositions. *Integr. Comput.-Aided Eng.* **2003**, *10*, 177–189. [CrossRef]
11. Hvam, L.; Mortensen, N.H.; Riis, J. *Product Customization*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2008.
12. Barták, R.; Salido, M.A.; Rossi, F. Constraint satisfaction techniques in planning and scheduling. *J. Intell. Manuf.* **2010**, *21*, 5–15. [CrossRef]
13. Sabin, D.; Weigel, R. Product configuration frameworks-a survey. *IEEE Intell. Syst.* **1998**, *13*, 42–49. [CrossRef]
14. Orsvärn, K.; Bennick, M.H. Use of Tacton configurator at FLSmidth. In *Knowledge-Based Configuration—From Research to Business Cases*; Felfernig, A., Hotz, L., Bagley, C., Tiihonen, J., Eds.; Morgan Kaufmann: Waltham, MA, USA, 2014; pp. 211–218.
15. Schäffer, E.; Shafiee, S.; Mayr, A.; Franke, J. A strategic approach to improve the development of use-oriented knowledge-based engineering configurators (KBEC). *Procedia CIRP* **2021**, *96*, 219–224. [CrossRef]
16. Gembarski, P.C.; Li, H.; Lachmayer, R. KBE-Modeling Techniques in Standard CAD-Systems: Case Study—Autodesk Inventor Professional. In *Managing Complexity*; Bellemare, J., Carrier, S., Nielsen, K., Piller, F., Eds.; Springer: Cham, Switzerland, 2017. [CrossRef]
17. Hirz, M.; Dietrich, W.; Gfrerrer, A.; Lang, J. *Integrated Computer-Aided Design in Automotive Development*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2013.
18. Skarka, W. Application of MOKA methodology in generative model creation using CATIA. *Eng. Appl. Artif. Intell.* **2007**, *20*, 677–690. [CrossRef]
19. Furferi, R.; Governi, L.; Uccheddu, F.; Volpe, Y. Computer-aided design tool for GT ventilation system ductworks. *Comput. Aided Des. Appl.* **2018**, *15*, 170–179. [CrossRef]

20. Milton, N. *Knowledge Technologies*; Polimetrica, S.a.s.: Monza, Italy, 2008; Volume 3.
21. Cunis, R.; Günter, A.; Strecker, H. *Das PLAKON-Buch: Ein Expertensystemkern für Planungs-und Konfigurierungs-Aufgaben in Technischen Domänen*; Springer: Berlin/Heidelberg, Germany, 1991.
22. Schröder, C.; Möller, R.; Lutz, C. A partial logical reconstruction of PLAKON/KONWERK. In Proceedings of the Workshop on Knowledge Representation and Configuration WRKP, Saarbrucken, Germany, 30 August 1996; pp. 55–64. [CrossRef]
23. LaRocca, G. Knowledge based Engineering: Between AI and CAD. Review of a Language based Technology to Support Engineering Design. *Adv. Eng. Inf.* **2012**, *26*, 159–179. [CrossRef]
24. Brailsford, S.C.; Potts, C.N.; Smith, B.M. Constraint satisfaction problems: Algorithms and applications. *Eur. J. Oper. Res.* **1999**, *119*, 557–581. [CrossRef]
25. Kumar, V. Algorithms for constraint-satisfaction problems: A survey. *AI Mag.* **1992**, *13*, 32. [CrossRef]
26. Petrie, C.J. *Automated Configuration Problem Solving*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2012.
27. Juengst, W.E.; Heinrich, M. Using resource balancing to configure modular systems. *IEEE Intell. Syst.* **1998**, *13*, 50–58. [CrossRef]
28. Amadori, K. Geometry Based Design Automation: Applied to Aircraft Modelling and Optimization. Ph.D. Thesis, Linköping University, Linköping, Sweden, 2012.
29. Li, H.; Gembarski, P.C.; Lachmayer, R. Template-Based Design for Design Co-Creation. In Proceedings of the 5th International Conference on Design Creativity (ICDC2018), Bath, UK, 31 January–2 February 2018; pp. 387–394.
30. Aldanondo, M.; Vareilles, E. Configuration for mass customization: How to extend product configuration towards requirements and process configuration. *J. Intell. Manuf.* **2008**, *19*, 521–535. [CrossRef]
31. Pitiot, P.; Aldanondo, M.; Vareilles, E. Concurrent product configuration and process planning: Some optimization experimental results. *Comput. Ind.* **2014**, *65*, 610–621. [CrossRef]
32. Kloock-Schreiber, D.; Domarkas, L.; Gembarski, P.C.; Lachmayer, R. Enrichment of geometric CAD models for service configuration. In Proceedings of the 21st International Configuration Workshop, Hamburg, Germany, 21 September 2019; pp. 22–29. [CrossRef]