Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 44 (2016) 204 – 209

www.elsevier.com/locate/procedia

6th CIRP Conference on Assembly Technologies and Systems (CATS)

Configuration management through satisfiability

Bryan Pearce[a], Mary E. Kurz[a*], Keith Phelan[a], Joshua Summers[a],

Jörg Schulte[b], Wolfgang Dieminger [b], Kilian Funk[c]

[a]*Clemson University, Clemson, SC 29634, United States*
[b] *BMW, Spartanburg SC United States*
[c] *BMW, Munich Germany*
* Corresponding author. Tel.: +1-864-656-4652; *E-mail address:* mkurz@clemson.edu

Abstract

In a Build-to-Order environment, a configurator relays the taxonomy of customization choices to the customer, then translates these choices into a bill of materials. Configuration Management (aka Variety Management) of the system entails validating proposed changes to the policies that govern both configurator processes. We present a satisfiability approach to the problem, in which a suite of conflict classes are developed, representing potential configurator failure modes. Satisfiability logic routines test the potential presence of each conflict class if the proposed change is adopted, using an integrated constraint set including both part allocation and customization object relationships.
© 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the organizing committee of the 6th CIRP Conference on Assembly Technologies and Systems (CATS)
Keywords: Configuration Management, Variety Management; Satisfiability

1. Introduction

Over the last few decades, global markets for manufactured goods have increasingly offered customizable products flexible to customer preferences. To remain relevant in this shifting economy, manufacturers have focused on mass customization practices that support the increase in product variety while retaining high production volumes [1][2].

Configuration management is the process of constructing and managing a domain of product variety space that meets customer needs. There are several component activities under this umbrella, including assessing customer preferences and product variant capabilities, and identifying specific product configurations that meet demand [3]. These problems are particularly difficult when the degree of customization choices is high, as each configuration management problem grows exponentially in response to each additional customization choice. Product families are a common method for managing this complexity, in which the domain of product variety space is divided into an array of independent base product platforms, each of which can be modified by the addition, subtraction, or substitution of modular options [4]. Even with a product family approach, however, maintaining product variety information remains a challenge for highly customizable products [5].

Build-to-Order production allows customers to configure their purchase personally, choosing from the domain of offerings made by the manufacturer. If product variety is small, with relatively few configuration alternatives, then the customer may be presented with a catalog enumerating each fully-configured offering. If product variety is high, however, an enumeration approach is not possible. The German automotive industry presents a compelling example, where

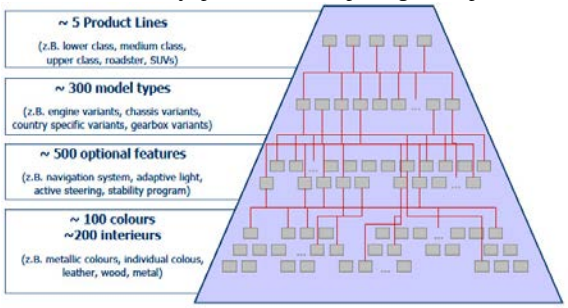

Fig. 1. Model of configuration variants in the automotive industry [7]

2212-8271 © 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).
Peer-review under responsibility of the organizing committee of the 6th CIRP Conference on Assembly Technologies and Systems (CATS)
doi:10.1016/j.procir.2016.02.127

vehicle configuration is a composite of many smaller subsystem configuration problems, such as color of paint, engine size, trim and badge options, etc. Taken as a whole, up to 10^{32} unique offerings may exist for some vehicles [6]. Fig. 1 shows an example of configuration complexity in the automotive industry. This diagram depicts the vehicle configuration problem hierarchically according to the product family approach, with platform at the topmost level and more detailed configuration features below.

A configurator is a software alternative to catalogs that divides the product configuration process into stages. In the first stage the customer selects the platform, or base product. In successive stages the customer is queried over a set of options corresponding to one particular subsystem of the product. For example, a vehicle configurator may first query for the model (tagged name), then later query for engine and drivetrain options, interior cabin options, etc. Once the customer completes and submits an order, the configurator constructs a bill of materials (BOM) by translating configuration choices into a set of corresponding parts.

The manufacturer requires control over the choices rendered by the configurator at each stage, to ensure that final configured product conforms to technological or marketing specifications. To this end, a rule-based reasoning technique guides the configuration process. Rule-based reasoning encodes a set of production rules, or constraint relationship between options, usually expressed as conditions and consequences relating options (if “A” then “B”). The configurator consults these production rules between each stage, checks whether previous customer choices meet the conditions of each rule, then constructs the next configuration stage to only include those choices that satisfy all rules. Another set of rules manages the mapping of the configured product to part allocation. Examples of rule-based configurators are given in [8][9][10].

Technology and market conditions change through time, inducing changes in product offerings, parts usage, or both. In operational terms, this entails making changes to the rule sets that control the configurator customer interface, and the part allocation processes that creates the BOM. The configuration management task is to validate a candidate set of rule alterations, to ensure that the alterations map correctly to the intended change and don’t create unintended side effects, e.g. an incorrect BOM for some product configuration.

The purpose of this paper is to develop support methods for this validation process, based on needs identified in previously published research. Section 2 summarizes the particular motivating scenario. Section 3 provides the background for the resulting validation process, which is described in Section 4, with examples. Section 5 concludes the paper.

2. Motivation

A wide variety of configuration management techniques have been developed to assist in the implementation. Phelan et al [11] describe several methods as well as potential challenges, which directly motivates the solution proposed here. The remainder of this section summarizes the findings of Phelan et al [11] for the convenience of the reader.

The foundation for the manufacturer’s configuration management system is a rule database that contains the rules governing the possible options and packages for a specific vehicle, resulting in a rule-based configuration management system. Production rules can be described as a set of conditions and consequences (if “A” then “B”). Therefore, the condition relates to an existing component or state of the product which, if met, results in an execution of the consequence action. An example of this would be as follows: “If Part A is found in the configuration, then Part B cannot be used in this configuration”. The scope of the rule database (over one thousand rules per vehicle) makes it difficult to ensure the accuracy of all of the rules and to ensure that the rule database covers the complete set of feasible configurations for each vehicle. Additionally, maintaining the rule database, with either updates or changes, is equally challenging due to the amount of possible change propagation and ensuring that all necessary changes have been made.

The rule database is used for at least three separate functions in the company. First, it is used for the ordering of vehicles which are all specified external from the manufacturing site, either by a customer or a dealership. Each vehicle built results from a selection of the possible components or options that are available or feasible based on location and other specified options. The tool used for specifying the vehicles relies on the above rule database. Second, the rules are used for part-ordering. Once a vehicle has been ordered, a parts management system uses the specified options to identify the parts that are required (and therefore ordered from suppliers) for the vehicle. Third, the line balancing utilizes the rules to accurately predict the time utilized for each worker and station. Tasks that cannot occur on the same vehicle do not contribute to the takt time and are detected by “violations” of the rules in the database; the larger of these task times should be used as the time for the set of those tasks. As all of the systems rely on the rule database, it is imperative that all of the rules are accurate and complete.

The rule database is updated throughout time based on marketing or engineering changes. Phelan et al report that much of the verification process for rule change is based on individual employee experience. For example, one employee reported that his experience with different vehicle systems has taught him to examine some areas more than others. Such reports were typical in the case study. However, this type of human verification is not feasible due to the scope of the rule set. There are approximately 1,500 parts per vehicle, with nearly 10 variants per component. Additionally, there are a half dozen models with dozens of variants and scores of options in configuring these components. Ultimately, there are nearly 700 million possible configurations that must be checked for feasibility periodically.

Over the course of the case study, the researchers identified numerous opportunities for improvement, which highlights the need for the work reported here. These are classified as follows, a few with an example of a motivating scenario.

- “Rule conflict.” Is there a subset of two or more rules such that no possible configuration may satisfy them?
- “Object activation.” Can all options/parts/etc. that are declared as being available for selection actually be selected?

- "Part family allocation." For a given family of alternative parts (e.g. all windshields), will one (and only one) of the parts be allocated for every configuration? A vehicle was being assembled for which there was no windshield. The selected options result in a configuration that is not feasible but this was not noted until the vehicle was in production. The correct allocation of part types should always result from each feasible vehicle configuration.
- "Part family matching." Consider a suite of several part families, some of which are intended to match to others for geometry or color reasons. Are the rules correctly implemented, or is there a configuration that mismatches parts? An exhaust system used with the diesel versions of a particular model was determined (in assembly) to be incompatible with a sports package due to a geometric constraint with the included fog lights. A rule was created to forbid the sports package and the specific exhaust system on the same vehicle, preventing such errors in the future. Months later, it was decided that the fog lights should be unbundled from the sports package. Unfortunately, the same production error resulted (exhaust system and fog light physical conflict) because the rule was not carried over from the sports package to the fog lights.
- "Antecedent satisfiability." Are there any rules for which the antecedent (IF- part) of the rule cannot be satisfied? If so, then the effects of the rule are inconsequential, as the rule is never active. This is a desirable test for rule database pruning.
- "Implicit relationships." Are there any binary inclusion/exclusion object relationships that are implicitly enforced, through the collected effects of explicit constraints? This was observed when a new rule was created (and approved) which artificially limited the possible windshield options for a given model. The added rule disallowed the selection of the option for an anti-glare strip on the windshield for certain models. However, due to limitations with the parts, this meant that the only allowable configuration for customers desiring the anti-glare strip would also be required to purchase the heads-up display option. There is no reason for the two options to require the presence of the other option but the two options were implicitly linked.

3. Rule structure and object ontology

Several different types of logical objects play a role in the configurator processes. Although all objects are Boolean in nature, being either present (true) or absent (false) on any final product, there exist varying ontological relationships both within and between object types. Quickly summarized, the five object types are:

- Options, conceptual optional modular features.
- Classes, sets of mutually exclusive option alternatives related to a single product feature.
- Packages, aggregated sets of options.
- Parts, the physical components assembled into the final product.
- Part families, sets of alternative parts for a single purpose, only one of which may be used per product.

Options and parts are atomic objects. Parts are physical entities as well as configuration objects, as part variables correspond one-to-one with inventory items that may be installed on the product. Options are conceptual representations of elemental configurable features, e.g. roof racks on a vehicle. The customer may either select an option or not, to express their preference for each feature. The other three object types, other than options and parts, are composite objects built from multiple option or part objects.

Package objects are marketing-derived batches of options. Packages are designed to allow a customer to quickly select a large array of synergistic options simultaneously, rather than individually choosing each individual option. For example, the advanced electronics package on a vehicle might specify selecting the heads-up display, backseat television screens, and one of two top end stereo options (but not any lower-tier stereos). Each of these options could be selected individually by the customer, if the package is not selected. The configurator presents packages to the customer at the start of each subsystem configuration stage, to permit quick selection of options for that subsystem.

Classes are used for product features that might intuitively be thought of as a single variable with multiple possible values. Rather than encoding such cases with a single multi-valued option variable, instead each value is modeled as a separate Boolean option object, and a class object is introduced that contains all the member options. For example, the "color" class may contain member red, yellow, and black Boolean option variables. The ontology of a class object requires that all members are mutually exclusive, allowing no more than one member to be present on any configured product. Class objects are also Boolean variables, and are true if some member option is true, and false if all member options are false. Classes are not exposed to the customer in the configurator, as they are intended to help maintain the integrity of production rules.

Part families are collections of alternative, mutually exclusive parts. Depending on the configuration selected, one of an array of modular parts is installed to perform a given function. Part families are similar to classes, except that part family members are always parts and classes include only options.

There are two types of production rules, corresponding to the two sequential configurator processes. The first set of rules governs interactions between options, classes, and packages, and is consulted by the configurator during the configuration process. These rules constrain the customer from configuring a product which is disallowed, by specifying the domain of the total configuration space that is permissible. The second set of rules governs the translation of a configured product into a BOM, by mapping each part to a set of conditions that call for the part. The conditions for each part are functions of the options and packages selected by the customer during the configuration process.

3.1. Background for Boolean satisfiability implementation

This section is included for completeness for the reader.

3.1.1. Rendering production rules as Boolean expressions

All object ontologies and production rules are reduced to first-order Boolean logic, a mathematical model and formal grammar used for reasoning about the truth of logical expressions. Boolean expressions are constructed as a series of operators and literals, assembled according to a formal grammar. Literals are Boolean objects, which are either true or false, and include all configuration objects discussed in section 3. Operators are the logical functions OR, AND, and NOT, used to conjoin literals into expressions. All configuration constraints encoded in the configurator customer interface and BOM translation subsystems are encoded as first-order Boolean expressions, and each expression must be satisfied by any valid configuration. Table 1 lists the components used to render production rules as Boolean expressions.

Table 1. Grammar of Boolean expressions

Expression component	Syntax	Example
AND	&	(A & B) means "both A and B"
OR	/	(A / B) means "A or B"
NOT	¬ or -	(-A) means "not A"
IMPLICATION	→	(A → B) means "if A, then B"
BICONDITIONAL	↔	(A ↔ B) means "A if and only if B"
Literal	<name>	A is itself

Production rules that are written in if-then form require no manipulation, as the $antecedent \rightarrow consequent$ format is already a Boolean expression. To this set of rules are appended additional Boolean expressions to reflect the relationships implied by class, part family, and package ontologies.

Each class object requires exactly one of its member options to be true. For each class, a Boolean expression is generated to represent membership, e.g. $Class \leftrightarrow mem_1/\ldots/mem_n$. Next, a set of Boolean expressions is generated to enforce exclusion constraints between members, taking the form $member_i \rightarrow \neg member_j, \forall i \neq j$. Each part family generates exclusion and membership expressions in the same way.

Packages are defined by expressions of the form P7S2A→ (!S255A) & (!S4CKA / S4ADA / S4B8A) . The antecedent of this implication is the package object, and the consequence is the set of options that defines the package. Note that the example consequence is clustered as a set of parenthetical clauses, each beginning with a "!" mark. Each of these clauses corresponds to a set of mutually exclusive options. After selecting a package, the configurator will prompt the customer to pick one option from each parenthetical clause. Each exclusive clause must be transformed to first-order Boolean logic. Consider the example clause $(!A / B / C)$. After transformation, the expression becomes $(A / B / C) \& -(A \& B) \& -(A \& C) \& -(B \& C)$.

3.1.2. Boolean Satisfiability

If all literals are assigned a truth value, then a Boolean expression containing them may be resolved to either true or false. If, on the other hand, some or all of the literals are unassigned, then the truth of the expression may be unresolved. Validating the set of production rules requires checking that all user-selectable configurations result in correctly specified, buildable products. When working with Boolean expressions that contain unspecified literals, a pertinent question may be "Is there any set of true/false values for literals that results in the expression resolving to true?" This question is known as the Boolean satisfiability problem (SAT).

A series of breakthroughs in the early 2000s spurred the development and proliferation of SAT solvers [12][13][14]. Industrial application of these tools to verification problems in railroad, avionics, and automotive sectors followed shortly thereafter [15][16]. These solvers take as input a set of Boolean expressions, and search for some combination of literal values that will "satisfy," i.e. result in all expressions evaluating true. If the solver finds that no possible combination of literal values can satisfy all expressions, then the solver returns UNSATISFIABLE. Otherwise, the solver returns SATISFIABLE along with the values found for each literal.

SAT solvers generally require inputs in the DIMACS conjunctive normal form (CNF) format. Each Boolean expressions of the form discussed in section 3.1 may be programmatically held in a binary parse tree data structure, populated by a recursive descent parse of rule strings. Nodes in a tree represent operators, with children nodes representing operands. Each binary tree is transformed to CNF by following these steps:

- Transform "!" Exclusive-OR subtrees
- Reduce binconditionals to two conditionals
- Reduce conditionals to AND, OR, NOT
- Propagate NOT downward towards leaves
- Distribute AND over OR

SAT problems do not strictly require that all literals be unspecified at the start. A partially specified SAT problem is accomplished by appending an additional expression for each literal that has an initial value. For example, appending the expression "-A" will force solutions where A=false.

The general strategy of the validation methods below is to construct a SAT problem instance (or suite of parallel SAT instances) that is (are) equivalent to a conflict. The SAT problem is then evaluated via the SAT solver. If the solver finds a solution, then this means that there exists a set of literal true/false values with which the conflict emerges.

4. Conflict Classes

The bulk of the contribution of this paper is presented in this section, in which the rule database is subjected to various tests based on which type of potential conflict is being considered. The following subsections describe independent experimental tests to check for potential conflicts arising after a change to production rules. The general methodology for each conflict detection method is to begin with all existing constraint information, aggregated across all production rules and object ontologies, as discussed in section 3. This union of production rules is referred to as the BASESAT herein. Each conflict detection method appends some test expression(s) to the BASESAT for the particular conflict detection class, then solves the SAT instance with a SAT solver. If a suite of tests is performed serially then all test expressions are removed between tests, resetting the BASESAT to its original state.

In each conflict class subsection below are details for constructing test expressions and logic for interpretation of SAT solver results, as well as example applications of each conflict class.

4.1.1. Rule conflict

A rule conflict test checks whether the change has created a conflict that prevents all rules from being satisfied simultaneously. The method to perform this test is to append the proposed change onto the BASESAT, and then apply the SAT solver. If the solver returns UNSATISFIABLE, then the change has created a conflict. If, on the other hand, the solver returns SATISFIABLE, this is not sufficient evidence to conclude that there are no problems with the change. There might still be issues that could be detected by performing some of the other conflict detection tests. For example, consider the existing rule on the first row of Table 2, and suppose the change is to induce the ruleTable 2 shown on row 2.

Table 2. Rule conflict example

Rule	Constraint
Current	L807A & S609A → S6AEA
Proposed	L807A & S609A → -S6AEA

Although this pair of rules is clearly nonsense, the solver will return SATISFIABLE if this experiment is performed. If either rule is "active," i.e. its antecedent is true, then the other rule will be violated. However, the apparent conflict can be avoided by the satisfiability routine, and a satisfying configuration found, if both rules are inactive. In this case, setting either option L807A or S609A to false will deactivate both rules. Adding this new rule would effectively forbid any valid configuration from having both L807A and S609A. See section 4.1.3 for testing conflicts of this type.

4.1.2. Object activation

Object activation tests iteratively check each individual object, to determine whether there exists a configuration for which the object is active. An object is disabled if no possible configuration activates the object. Disabled objects are evidence of errors arising from the changes. The types of objects considered may be divided into two categories, literals and constraints. The following subsections discuss these.

4.1.2.1. Literal activation

Literals include parts, options, packages, and class objects. Each literal is checked individually, to determine whether the literal is active on some configuration. A single test expression is appended to the BASESAT during each iteration, forcing the literal to be active, e.g. $\neg S323A \rightarrow S323A$. If the SAT solver returns SATISFIABLE then the literal may be active. Else, if UNSATISFIABLE is returned, then the literal is not permitted on any configuration. Disabled literals are evidence of errors in the implemented changes.

4.1.3. Antecedent Satisfiability

The antecedent is the "IF" portion of a rule. Testing antecedent satisfiability verifies that a rule can be activated. The scenario presented in section 4.1.1 showed a latent conflict, causing rule deactivation. In cases such as this, activating the rule results in no satisfying configurations. This test iteratively tests each rule, by appending a forcing expression for the rule to the BASESAT. For example, Table 3 shows an existing rule, and a corresponding test rule forcing it to be active. If the solver returns SATISFIABLE for this test, then the rule can be activated. Else, the proposed change has forced the rule to be inert, and is evidence of a potential error.

Table 3. Antecedent satisfiability test expression

Rule	Constraint
Current	S2D4A / S2H4A → S258A
Force	- S2D4A & - S2H4A → S2D4A / S2H4A

4.2. Implicit inclusion / exclusion

Implicit relationships between option pairs can be either an inclusion, if the options must always occur together, or an exclusion, if the options may never occur together. The test operates iteratively, checking each pair of options in turn.

For a given pair of options, inclusions are found by testing the contradiction, "Can one option be active, but not the other?" A pair of test expressions similar to those in Table 4 are appended to the BASESAT. If the solver returns UNSATISFIABLE, then the option pair has an inclusion relationship.

Table 4. Implicit inclusion test expressions

Rule	Constraint
Current	-S323A → S323A
Force	S4FFA → -S4FFA

Exclusions between the option pair are also found via contradiction, by testing the question, "Can both options be active simultaneously?" A pair of test expressions similar to those in Table 5 are appended to the BASESAT. If the solver returns UNSATISFIABLE then the option pair has an exclusion relationship.

Table 5. Testing implicit exclusion

Rule	Constraint
Force +	-S5DPA → S5DPA
Force +	-S645A → S645A

4.3. Part Family Allocation

The part family allocation test verifies whether exactly one of the parts in the family is allocated for every vehicle. This result is achieved in two stages, first testing whether zero of the parts may be allocated, then testing whether two or more of the parts may be allocated.

The first stage tests for contradiction, "Can all of the parts in the family be inactive?" A test expression similar to that in Table 6 is appended to the BASESAT, to force all parts in the family inactive. If the solver returns SATISFIABLE, then it is possible to build a vehicle using none of them. If the solver

returns UNSATISFIABLE then the stage 1 test is passed, and stage 2 tests begin.

Table 6. Part family activation

Rule	Constraint
Force -	7292394 / +7292399 → -7292394 & -7292399

The second stage tests whether more than one of the parts in the family may be allocated for any vehicle. This result is achieved iteratively, checking each part in the family in turn. A direct proof is offered by the question, "If part *i* (chosen by iteration) is active, can another part in the family be active as well?" A test expression similar to that in Table 7 is appended to the BASESAT, to force one of the other parts to be active along with part *i*. If the solver returns SATISFIABLE for any of the iterated tests, then a configuration has been identified that calls more than one of the parts. Else, if the solver returns UNSATISFIABLE, then the stage 2 test is passed for this iteration. If all iterations return UNSATISFIABLE, the stage 2 test is passed.

Table 7. Testing multiple part inclusion from one part family

Rule	Constraint
Force *i*	-7292394 → 7292394
Force another	7292394 → 7292399 / 7292400 / 7292401 / 7292402 / 7292403 / 7308905

4.4. Part Family Matching

Parts are commonly designed to fit with other specific parts. In this section scenarios are considered where parts from one family are designed to match with another family. The conflict tests whether mismatched parts may be found for any configured product.

Consider an example scenario with four different part families. Let two part families hold parts related to exhaust tips, one family for round profiles and one family for rectangular. Let the other two part families hold parts related to bumpers, with one family for round exhaust holes and one family for rectangular exhaust holes. Geometry constraints require matching round bumpers with round exhaust tips, and rectangular with rectangular.

A contradiction test is performed to ensure that mismatches cannot occur. A test expression similar to that in Table 8 is appended to the BASESAT, forcing one of the part families to be active, and forcing its matching family inactive.

Table 8. Testing part family matching

Rule	Constraint
Force *Round Bumper*	-(RND BUMPER) → (RND BUMPER)
Force Not *Round Exhaust*	(RND EXHAUST) → -(RND EXHAUST)

If the solver returns SATISFIABLE for this test, then a configuration has been identified with mismatched parts, suggested that there are errors in the part allocation rules that construct the BOM.

5. Conclusion

Changes to product offerings require updating a rule-based configurator's knowledge base to reflect the change. Validating these changes entails checking that only the desired product configurations are available to the customer, as well as verifying that configurations call the correct parts when constructing the BOM. This paper presents a set of conflict classes, each of which corresponds to a potential failure mode of the configurator, and a formalized SAT search procedure for each type of conflict.

References

[1] G. Da Silveira and D. Borensein, "Mass Customization: Literature review and research directions," *International Journal of Production Economics*, vol. 72, no. 1, pp. 1-13, 2001.

[2] L. F. Scavarda, A. Reichhart, S. Hamacher and M. Holweg, "Managing product variety in emerging markets," *International Journal of Operations & Production Management*, vol. 30, no. 2, pp. 205-224, 2010.

[3] Tidstam and J. Malmqvist, "Information Modelling for Automotive Configuration," in *NordDesign*, Göteborg, Sweden, 2010.

[4] J. R. Jiao, T. W. Simpson and Z. Siddique, "Product family design and platform-based product development," *Journal of Intelligent Manufacturing*, vol. 18, no. 1, pp. 5-29, 2007.

[5] R. S. Renu and G. M. Mocko, "Decision Support Systems for Assembly Line Planning: modular Subsystems for a largescale Production Management System," Clemson, SC, 2013.

[6] H. Meyr, "Supply Chain Planning in the German Automotive Industry," *OR Spectrum*, vol. 26, pp. 447-470, 2004.

[7] E. Knippel and A. Schulz, "Lessons learned from implementing configuration management within electrical/electronic development of an automotive OEM," in *Proceedings of 4th Annual Symposium of INCOSE*, Toulouse, France, 2004.

[8] Choi and S. Bae, "An Architecture for Active Product Configuration Management in Industrial Virtual Enterprises," *International Journal of Advanced Manufacturing Technology*, vol. 18, no. 2, pp. 133-139, 2001.

[9] H. Andersson, S. Steinkellner and H. Erlandsson, "Configuration Management of Models for Aircraft Simulation," in *Proceedings of the 27th International Congress of the Aeronautical Sciences*, 2010.

[10] R. Raffaeli, M. Mengoni, M. Germani and F. Mandorli, "An Approach to Support the Implementation of Product Configuration Tools," in *ASME Design Engineering Technical Conference*, San Diego, CA, 2009.

[11] K. Phelan, C. Wilson, J. D. Summers, M. E. Kurz, "A case study of configuration management methods in a major automotive OEM," in *Proceedings of the 2014 International Design Eningeering Technical Conference*, Buffalo NY USA, 2014 (DETC2014-34186).

[12] E. Goldberg and Y. Novikov, "A fast and robust sat-solver," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Paris, 2002.

[13] Marques-Silva and K. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEETC: IEEE Transactions on Computers*, vol. 48, 1999.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an efficient SAT solver," *DAC*, 2001.

[15] M. Penicka, "Formal approach to railway applications," *Formal Methods and Hybrid Real-Time Systems*, pp. 504-520, 2007.

[16] Hammarberg and S. Nadjm-Tehrani, "Formal verification of fault tolerance in safety-critical reconfigurabe modules," *Journal of Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 268-279, 2005.