Testing Product Configuration Knowledge Bases Declaratively

Konstantin Herud[1], Joachim Baumeister[1,2]

[1]*denkbares GmbH, Germany*
[2]*University of Würzburg, Germany*

Abstract

Product configuration typically makes use of declarative knowledge to model the properties of complex products. The development of such product knowledge bases is similar to the development of code bases. Key challenges include collaboration, maintainability, extensibility, and quality assurance. New features, requirements, and regulations lead to frequent and error-prone iterations. Analogous to software engineering, automated testing is critical to ensure the integrity of product knowledge. While the general NP-complete complexity of configuration problems generates much academic interest, these business-relevant challenges receive less attention. This paper thus presents ongoing work on quality assurance in product configuration using regression testing. Therefore we first formally define a novel data structure for performing the tests. We then explore the challenges of collaboratively engineering testing knowledge in practice. Finally, we illustrate a grammar to formulate the tests with several application scenarios.

Keywords

Product Configuration, Regression Testing, Declarative Knowledge, Quality Assurance, Collaboration, Maintainability

1. Motivation

Product configuration describes a broad area that deals with the composition and individualization of generic components. A typical example is the selection of a custom computer. Instead of offering a predefined set of options, for example, customers can instead assemble the exact computer they want from components such as different processors, cases, and monitors. As a result, the users are more likely to spend money. The various components are contained in a product catalog and are subject to certain compatibilities with each other. However, Felfernig et al. [1] shows that modern product configuration is also used for much more complex problems. Examples include railway interlocking systems [2], cement plants [3], mobile phone networks [4], offers, contracts, user manuals, or technical documentation [5], and services like elevator maintenance [6]. In such cases, simple compatibilities are not sufficient, since, for example, legal regulations, spatial and temporal requirements, or other physical conditions must be taken into account. Thus, manual modeling of the domain becomes necessary. Despite the long history and profitability of product configuration, there are still several issues that make engineering product knowledge a challenge.

LWDA'22: Lernen, Wissen, Daten, Analysen. October 05–07, 2022, Hildesheim, Germany
konstantin.herud@denkbares.com (K. Herud); joachim.baumeister@denkbares.com (J. Baumeister)

Some of these challenges are similar to software development. While executing code performantly is important, the real challenge is developing high-quality code in the first place and maintaining it over a long time. By having different parties making continuous changes to the code, bugs are easily introduced. The same applies to product configuration. However, configuration knowledge must be completely free of errors to prevent customers from ordering products that can neither be manufactured nor sold. A system for automated quality assessment of knowledge integrity is thus essential. One tool to identify newly introduced errors is regression testing. However, because NP complexity is typically associated with configuration problems, difficulties exist in developing meaningful tests at all. Unlike in software development, it is not sufficient to describe a set of expected inputs and outputs. Although expected inputs are usually well defined, their amount of possible combinations is exponential to the number of feature values. This quickly exceeds the capacity of hardware and developers. To address this, our work develops a novel data structure for regression testing in product configuration to deal with this complexity. Analogous to the development of product knowledge itself, declarative formulation is a guiding principle. Often, different parties with different expertise maintain the knowledge. These parties should not be concerned with the details of a procedural, and thus technical, formulation. Test knowledge is inherently declarative: It formulates *what* the desired behavior of functions is, rather than knowing their details of *how*. This notion leads us to the idea of test-driven knowledge development. New requirements are first formulated as tests and thereby documented simultaneously. Based on this, the requirement can be understood in the long term, and arbitrary refactoring can be performed on the knowledge.

This paper therefore addresses two questions:

- How can products be regressively tested despite an exponential set of possible configurations?
- How does the collaborative development and maintenance of test knowledge between parties of different expertise work?

Our work outlines a vision of a set of methods for quality assurance and collaborative development of complex product configuration knowledge bases.

For this purpose, we first briefly define product knowledge and the configuration process in general in Section 2. We then present our data structure for formulating test knowledge in Section 3. In the next Section 4, we take a look at how tests are developed in practice and their lifecycle. We illustrate this view with several application scenarios in Section 5. Finally, we conclude our work with a brief look at the related literature and an overview of future challenges.

2. Product Configuration

The main idea of product configuration is the feature-based personalization of a product. There are serveral efforts for a general ontology to model the knowledge about a product. Since the set of possible configurations grows exponentially with the amount of selectable options, it is not feasible to enumerate all results in a database. Instead, configuration problems are commonly modeled as constraint satisfaction problems. Here, a set of variables and their domains exists, i. e., components and their type, or in general, features of a product. This typically involves the notion of customizing systems out of generic components that form a part-of-hierarchy. We refer to this as *structural knowledge*. Furthermore, the interactions of the variables are modeled with constraints. We subsequently describe this as *behavioral knowledge*. Finally, we define the configuration process itself. Common extensions of a general ontology consist of a hierarchy and dynamic activation of variables. More complex components are composed in this way recursively from more specific components which in turn are configurable. Since this hierarchical modeling introduces much complexity with concepts such as partonomy and taxonomy that are beyond the scope of this work, we stick with the notion of *features*, i. e., configurable aspects of the product. To avoid the irrelevant overhead of this complexity, we keep our definition as simple as possible to provide the basis for regression testing. We align ourselves with the definitions in [7].

Definition 1 (Feature)**.** The properties of a product are specified by n *features*. A feature f is a variable defined by its *feature type*. The *type* of a feature $domain\,(f)$ determines its non-empty domain. *Primitive* feature types describe numeric, boolean, and textual domains. *Concrete* feature types define a discrete and finite domain of selectable options.

We use typewriter font for concrete examples and write features with a capital initial letter. Domain values are completely capitalized. For example, the color of a car could be described by a feature Color, where $domain\,(\texttt{Color}) = \{\textsc{black},\ \textsc{white},\ \textsc{red}\}$. One reason for using the term feature is to distinguish between different types that capture different characteristics of a product. For example, there are also primitive data types that cannot be classified under the term component. Note that this definition can give rise to a potentially infinite number of possible product configurations, e. g., by having a feature with $domain\,(f) = \mathbb{R}$. The implication is that in this case no finite set of configuration exists. Thus no finite set of classical test cases with expected input and output can cover the entire space of valid configurations.

Since the individual configurations cannot be explicitly enumerated, they are implicitly specified by constraints. With the previous definition, only the configurable dimensions of a product are defined. However, a large part of the knowledge is formed by the constraints on these dimensions. We call these constraints *behavioral knowledge*. The set of all *possible* configurations results from the cross product of the domains of all features. However, usually only a fraction of them form *valid* configurations, i.e., combinations of the components that can be produced and sold. To capture the set of all valid solutions in this space of all possible solutions, constraints are required.

Definition 2 (Constraint)**.** A *constraint* c is a function that maps a configuration X to a boolean truth value, i. e.,

$$c : X \to \{\top, \bot\}\,. \tag{1}$$

While the configuration will be discussed in more detail in a moment, in our work we are mainly interested in first order and propositional as well as arithmetic constraints. Although the type of constraints is unbounded in theory, usually there are common patterns to describe the behavior of a product. Note that we abuse notation here to describe constraints c as formulas for which an interpretation I exists such that $I(c)$ evaluates to true or false.

1. Allowed or forbidden value *combinations* of different components, e. g.,

$$\begin{aligned}&(\texttt{Body} = \texttt{CITY} &&\wedge\ \texttt{Drive} = \texttt{FRONT\_WHEEL})\\ \vee\ &(\texttt{Body} = \texttt{SPORT} &&\wedge\ \texttt{Drive} = \texttt{REAR\_WHEEL})\\ \vee\ &(\texttt{Body} = \texttt{OFF\_ROAD} &&\wedge\ \texttt{Drive} = \texttt{ALL\_WHEEL})\end{aligned}$$

   This pattern lists possible feature combinations in table-like disjunctive normal form.
2. *Requirements*, that formulate arbitrary conditions that have to be fulfilled, e. g.,

$$\texttt{WeightInKG} <= 3500$$

   Although these are in nature similar to combinations, they formulate more concise constraints that go beyond (in)equality. They may involve arithmetic, e. g., the "weight of of all components must not exceed a certain value".
3. *Implications* which are similar to requirements. Here, a condition must first occur before the consequence must be met, e. g.,

$$\texttt{Body} = \texttt{SUV} \rightarrow \texttt{HorsePower} >= 100$$

Other examples outside the scope of this work describe default value assignments, involve temporal conditions, or concern the presence and absence of components in the hierarchy. The combination of features, feature types, and constraints of a product forms a *knowledge base*.

Definition 3 (Knowledge Base)**.** A *configuration knowledge base* is a triple (F, D, C), where F is the set of all features, D is the set of all feature types, and C is a set of constraints over F.

The knowledge base is then used to offer individual *configurations* to customers.

Definition 4 (Configuration)**.** A *configuration* X is a set of at most one value assignment x to each feature $f \in F$ in a knowledge base (F, D, C).

$$X = \{f = x \mid f \in F \wedge x \in \mathit{domain}(f)\}. \tag{2}$$

X is *complete*, if (3) holds, and *valid* if (4) holds.

$$\mathit{complete}(X) : \forall f \in F : \exists x \in \mathit{domain}(f) : (f = x) \in X \tag{3}$$

$$\mathit{valid}(X) : \forall c \in C : c(X) = \top \tag{4}$$

A complete configuration is therefore merely the finished process of feature binding where exactly one value exists for each feature. The *user requirements* U are a partial configuration, where each feature assignment $f = x$ is explicitly given by the user. Neither $\mathit{complete}(U)$ nor $\mathit{valid}(U)$ have to be true. We are interested in the set of solutions S, where each solution $s \in S$ is complete and valid. Note that $s \supseteq U$, since it is completed from U. We refer to U as *satisfiable* if $S \neq \emptyset$ and *unsatisfiable* otherwise. The configuration process is typically a sequence where U grows incrementally. The customer starts with $U = \emptyset$ and can select only valid or arbitrary values depending on the environment.

3. Regression Testing

Regression testing is commonly understood as the repeated evaluation of test cases to ensure that modifications in already tested functions do not cause new errors [8]. These new errors can arise, for example, from fixing old errors, refactoring in general, or by implementing new requirements and regulations. The term *regression* is used when a new version does not correctly maintain existing functionality. For the purpose of identifying these regressions, *test cases* are implemented. In product configuration, a strict separation of concern for testing exists in our work. Testing is for quality assurance of the knowledge base, not the configuration environment or a reasoning engine. By reasoning engine we refer to a system that is able to infer logical consequences from a set of asserted axioms and facts. For example, a common choice is Answer Set Programming (ASP) [9]. Here the axioms are first-order logical rules — the so-called problem encoding that is used to guide the search. The facts are ground atoms, which describe a problem instance. These instantiate propositional rules with which the problem is ultimately solved. However, a central idea in product configuration is the clear separation between knowledge modeling and reasoning engine. It is inefficient to address all possible reasoning tasks with a single solving technique, such as providing explanations in the case of failure, enumeration of models, optimization, or continuous value computations [10]. A decoupling of modeling and reasoning is therefore essential. In this work, we assume that there exists an arbitrary but correctly working reasoning engine with proper axioms. The goal is thus to ensure that the knowledge accurately models the product. This means that the solution space S only allows valid configurations and that customers cannot order invalid ones. A special requirement resulting from this is that not a single test is allowed to fail.

Although various synergies exist between knowledge and software engineering, there is a key difference for regression testing in product configuration. While it is usually possible to develop tests to cover the entire software, NP complexity generally prevails for constraint-level configuration problems [11]. This complexity makes it difficult to develop dedicated tests for the exponential set of allowed and disallowed configurations. Therefore, it is not sufficient to define a set of valid and invalid user requirements as test cases that are expected to be satisfiable or unsatisfiable. Instead, we extend the expressive power of the tests to the power of the configuration ontology itself. This means that the tests are fomulated declaratively as constraints — equivalently to the development of the product knowledge itself.

Definition 5 (Test Case)**.** A *test case* is a triple (U, t, m), where U is a potentially empty set of feature assignments $f = x : f \in F \wedge x \in \mathit{domain}\,(f)$, t is a constraint following Definition 2, and $m \in \{\mathit{universal}, \mathit{existential}\}$ is a reasoning mode.

A novelty here is the introduction of a reasoning mode m. The two modes $\mathit{universal}$ and $\mathit{existential}$ specify whether the test constraint must hold in at least one or all of the configurations that can be derived from the user requirements. Instead of developing and executing separate test cases for each possible configuration, efficient algorithms like CDCL can then be used to enumerate models [12]. The test constraint can thus be evaluated quickly in practice despite the exponential amount of configurations.

Definition 6 (Universal Testing)**.** Given a test case $(U, t, \mathit{universal})$, the test constraint t must hold in all configurations $s \supseteq U : \mathit{valid}(s) \wedge \mathit{complete}(s)$.

$$\forall s \in S : t(s) = \top \tag{5}$$

This type of reasoning is necessary for both positive and negative hard requirements. Section 5 will illustrate both reasoning modes in more detail.

Definition 7 (Existential Testing)**.** Given a test case $(U, t, \mathit{existential})$, at least one configuration $s \supseteq U : \mathit{valid}(s) \wedge \mathit{complete}(s)$ must exist in which the test constraint t holds.

$$\exists s \in S : t(s) = \top \tag{6}$$

This type of reasoning is for example important for a guided configuration process, where the user is pointed to the set of selectable valid values. Thus, it would be conceivable to implement concrete features as a dropdown menu, where the selectable options are loaded dynamically depending on the current requirements. A test case could then ensure that an option is still available. Both modes can be negated by negating the test constraint and using the opposite reasoning mode. The opposite of universal testing thus changes from $\neg\forall s \in S : c(s) = \top$ to $\exists s \in S : c(s) = \bot$. Correspondingly, existential testing changes from $\neg\exists s \in S : c(s) = \top$ to $\forall s \in S : c(s) = \bot$. The default mode tests the constraint accordingly with a configuration that complies with all satisfiable default rules. All feature assignments not derived from defaults can have a non-deterministic choice in this case. However, before discussing these modes in more detail, we define our collected test knowledge as *test suite*.

Definition 8 (Test Suite)**.** A *test suite* is a tuple (M, T), where M is a knowledge base following Definition 3 and T is a finite set of test cases following Definition 5.

The tests of a test suite can be executed sequentially by a portfolio of reasoning engines, but can also be fully parallelized. Since each test case is potentially associated with an NP complete evaluation, the performance of the test system plays a critical role. In addition, optimization can be performed, such as grouping several test cases with the same user requirements and the same reasoning mode.

According to Junker [13], the configuration task also consists of an explanation of failure if no configuration can be found that satisfies all requirements. Thus, further reasoning modes would be conceivable, for example to reason about properties of unsatisfiable configurations. Another mode could be used to test default assignments if they are supported by the ontology. Here, defaults are a separate set of constraints according to the scheme

$$\mathit{constraint} \rightarrow f = x \tag{7}$$

where $f \in F$ and $x \in \mathit{domain}(f)$. An example is $\texttt{Body} = \textsc{sport} \rightarrow \texttt{Seats} = \textsc{sport}$. Finding a configuration, for example, can then additionally be treated as an optimization problem to satisfy as many defaults as possible. Besides the purpose of grouping functionally similar options, defaults also serve to determinize the reasoning process. However, additional reasoning modes remain open as future work.

4. Engineering Testing Knowledge

Having formally defined both product configuration and regression testing, in this section we take a look at it from a practical standpoint. For this, we outline in Section 4.1 the collaborative development of test knowledge and its challenges using a domain-specific grammar. Then, in Section 4.2, we describe the lifecycle of the test knowledge, its integration into the development process, and most importantly, the execution of the tests.

4.1. Development

Various front ends are conceivable for developing the testing knowledge and thus setting up the data structure in Definition 8. These front ends must be adapted to the expected developers of the tests. Ideally, the front end is based on existing technology for developing product knowledge. Very technical developers, for example, could be granted direct access to a programming interface. Technically-averse actors, for instance subject matter experts, should be supported in other ways. One way would be to add a developer mode to the configuration environment. On the one hand, this allows to manually create a configuration to be tested analogous to the customer experience. On the other hand, also test constraints can be created in the developer view for each step, for example with the support of graphical editors and pre-built templates. The respective steps of the development process are then serialized into individual test cases. While this method is very convenient, it also comes with disadvantages. The linear navigation through the configuration process leads to a similar set of user requirements and thus similar test cases. This in turn leads to the many dimensions of the solution space of valid configurations possibly being insufficiently tested, while few other dimensions are redundantly tested.

A third solution would be a middle ground, such as the implementation of a Domain Specific Language (DSL). Technical details such as the different reasoning modes should be abstracted. A strongly declarative solution could be closely oriented to natural language. A DSL has the advantage of being able to use the synergies with text-based software development. This makes it very easy, for example, to integrate versioning with helpful difference views. Furthermore, a semantic wiki can serve as a platform for collaborative development, structuring and maintenance of test knowledge. In addition, text is a suitable interface for pointing out anomalies, for example through syntax highlighting. Ultimately, there is an integrated development environment (IDE) that combines all of these functions. The basis for this is the DSL. Such a language is briefly outlined in Listing 1.

Here, we only show the formulation of the constraints. As lines 1–3 show, a statement begins with the reasoning mode. Then, after lines 5–11, a constraint is initially a logical expression that can be evaluated to true or false according to Definition 2. Note the precedence of the rules. Essential for this is the `compare` rule in line 10 to evaluate constants obtained from formulas. Formulas in lines 13–23 represent calculations and functions that can be evaluated to constants. The last line refers to the hidden `feature` rule, which is used to query configuration-dependent values. Since typically different people without a computer science background develop tests, one goal of the DSL is to avoid computer science-specific concepts and terminology. For example, “&&” and “||” often serve as logical AND and OR in programming languages, but are replaced here by their natural language counterparts. Also, Unicode symbols are avoided, such as “$\rightarrow$”, “$\vee$”,

and "∧", which are inconvenient to type. Note that the grammar lacks operations to manipulate the configuration. However, a simple option would be to specify a serialized configuration with feature assignments for each test case, which is then loaded from a database or file. A self-contained test case, on the other hand, would start from a blank configuration and include operations to set and modify values. The written DSL code is then decomposed into a set of user requirements, where each element of the set arises after an operation to modify a value. We show examples of usage in Section 5.

```
1            test := univeral | existential
2        univeral := 'require' constraint
3     existential := 'allow' constraint
4
5      constraint := implication
6     implication := disjunction ('implies' disjunction)*
7     disjunction := conjunction ('or' conjunction)*
8     conjunction := negation ('and' negation)*
9        negation := 'not' negation | '(' constraint ')' | compare
10        compare := formula (operator formula)+
11       operator := '<' | '<=' | '>' | '>=' | '=' | '!='
12
13        formula := addition
14       addition := subtraction ('+' subtraction)*
15    subtraction := multiplication ('-' multiplication)*
16 multiplication := division ('*' division)*
17       division := sign ('/' sign)*
18           sign := '-' sign | '+' sign | other
19
20          other := '(' formula ')' | aggregation | atom
21    aggregation := function '(' formula (',' formula)* ')'
22       function := 'count' | 'sum' | 'min' | 'max' | 'equal'
23           atom := 'true' | 'false' | number | feature
```

Listing 1: A simplified grammar to define test cases. For simplicity, we adapt the symbols $\{(,), ?, *, +\}$ of regular expressions with the same meaning. The grammar is not functional due to a lack of terminal rules. Also, whitespaces are ignored. The terminal symbol `feature` is not defined, but is used to identify features $f \in F$. Likewise, `number` is not defined, but describes integer and floating point values. The start rule is `test`.

4.2. Execution

To detect regressions, tests must be run automatically after changes and new versions. Here, software development can serve as inspiration. Correspondingly, a *continuous integration* (CI) system can be implemented that runs the tests after each change or after manual triggering [14]. Figure 4.2 shows a prototypical implementation for the developed test framework of this paper.

The automated tests assert the integrity of new knowledge before it is accepted as the central consensus. Then, if *all* tests are successful, a resulting artifact can be delivered, for example. An artifact might be a compiled text file in a format that is ready for a reasoning engine. Following this idea, the concept of test-driven development can be adapted to product configuration. In order to integrate new features, requirements or regulations into the product knowledge, test cases are first developed for them. The lifecycle of product knowledge thus consists of iterations of the following steps [15]:

1. Clear definition and documentation of the new requirements
2. Formulation of the defined requirements as a test case
3. The implemented test fails expectably but not necessarily
4. Initial formulation of the new product knowledge
5. The implemented tests are now succesful
6. Refactoring of the existing knowledge

It is often easier to formulate conditions of validity than to formulate the knowledge itself. For example, it is easier to specify that the weight of a product must lie within a certain interval than to specify the calculation behind it. Another big advantage is that any requirements for the product are clearly documented. This is crucial in the case of regressions. Tests can fail for various reasons. In the simplest case, for example, the names of features change. If the configuration ontology contains a hierarchy, the position of knowledge in the hierarchy may change. It may also happen that old knowledge is invalidated for example by new regulations Therefore, beside the product knowledge also the test knowledge must be maintained. Both is however only possible, if the purpose of the knowledge can be comprehended afterwards by people other than the original developers. Good software is often self-explanatory. The origin and purpose of knowledge is not necessarily so. If there is ambiguity about whether old functions are obsolete in case of regression, then legacy data accumulates which has a strong negative impact on the quality of the knowledge base.

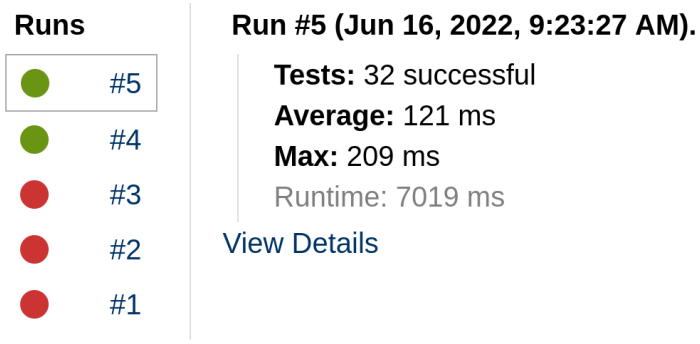

Figure 1: A screenshot of a prototypical implementation for the continuous integration of knowledge changes in the semantic wiki *KnowWE* [16]. At a defined trigger all regression tests are executed before the knowledge base is delivered. An overview shows the last executions and their status of success. In addition, quick information about the test execution is displayed, e. g., how many tests were executed, how long the test duration was and how long the runtime of the test process was including the setup of the configuration inputs. If required, further details on individual tests can be viewed.

5. Application Scenarios

Regression testing is used by companies to verify that the solution space of sellable configurations is correctly defined. We separate this interest into two categories:

1. **Whitelist Testing:** On the one hand, the set of manufacturable configurations should be clearly defined in order to prevent profit losses by selling fewer products due to a lack of configurations.
2. **Blacklist Testing:** On the other hand, non-manufacturable configurations are to be prevented in order to avoid problems with the correction of erroneous orders in downstream systems such as supply chain management and more importantly, the production line.

Since whitelisting explicitly describes allowed configurations, knowledge is much easier to maintain. This works, for instance, by describing allowed instead of forbidden feature combinations. For example, the list in Section 2 shows allowed combinations. The opposite — forbidden combinations — would be a negation of the entire expression. This would hide which configurations are excluded by the constraint. The situation is similar with test knowledge. It is hard to test the unknown unknowns. Whitelisting is therefore the recommended approach.

5.1. Whitelist Testing

Whitelist testing is concerned with the known properties of valid configurations. In the easiest case, tests are developed for an existing product whose knowledge base is merely being expanded or maintained. In this case, test cases can be generated automatically, which, for example, ensure that the sold configurations of a past period are still sellable with the revisions. A typical test case here consists of the user requirements U at that time and the feature assignments of the sold configuration. U is then universally tested, since the requirements come explicitly from the user and thus must be included in any derived configuration. The assignments of the reasoning engine A at that time are existentially tested to ensure that the configuration could still be derived in exactly the same way. For example,

$$U = \{\texttt{Body} = \texttt{SPORT}, \texttt{ExteriorColor} = \texttt{RED}, \texttt{Wheels} = \texttt{21\_INCH\_SPIDER}\},$$
$$A = \{\texttt{Seats} = \texttt{SPORT}, \texttt{InteriorColor} = \texttt{BLACK}, \dots\}.$$

A test case may then look like the following.

```
// load / setup configuration
require Body          = SPORT
require ExteriorColor = RED
require Wheels        = 21_INCH_SPIDER
allow   Seats         = SPORT AND
        InteriorColor = BLACK
...
```

Since the tests are generated automatically, each line can easily consist of a single constraint. Otherwise, universal constraints can be linked with a logical AND to save typing. Note that for existential tests, it makes a difference, though, whether they are formulated as a single or a

separate constraint. If the ontology supports defaults, then a third reasoning mode for it would be conceivable, that explicitly tests configurations that maximize all satisfiable defaults (see Section 3). For example, if `Body = SPORT` the implication `Seats = SPORT` should be tested as default instead of existentially.

However, usually it is not old knowledge that needs to be tested, but new knowledge. Instead of a set of inputs and expected outputs, the requirements of the new knowledge must then be specified as testable criteria. For example, one property of any product that is often part of the knowledge base is its price calculation. Here, prices are often dynamically composed of surcharges and price reductions. A surcharge could result if a product does not use a uniform color, but different colors for different components. This requirement is first formulated as a test.

```
require equal(ExteriorColor, InteriorColor, CoverColor)
        implies ColorSurcharge = false

require not equal(ExteriorColor, InteriorColor, CoverColor)
        implies ColorSurcharge = true
```

The test starts with $U = \emptyset$, since it must apply universally to all derivable configurations. Then the functionality is implemented in the product knowledge itself and can be refactored as desired. Note that the derivation of `ColorSurcharge` here is potentially very similar to the test constraint itself. However, this redundancy documents the original requirement of the product knowledge in the event that the knowledge is changed.

5.2. Blacklist Testing

Blacklist testing is concerned with the unknown properties of invalid configurations. Because of this uncertainty, blacklist tests are discouraged. However, they are often required by poor knowledge modeling practices. For example, by using combination constraints (see Section 2) to define forbidden combinations instead of allowed ones. Equivalently to tests with sold configurations, the reasoning engine can be used to generate a set of non-manufacturable configurations. To do this, U is randomly generated repeatedly and $\neg valid\,(U)$ is checked in each case. However, the set of invalid configurations typically exceeds the set of valid configurations by several orders of magnitude. This severely limits the quality assurance of these blacklist tests. Much better suited are universal tests with general statements, where $U = \emptyset$. As an example, consider again the price. A universal statement would be, for example, that the price must always lie in an interval that is statically calculated from the minimum and maximum prices of the components.

```
require 12,517.32 <= sum(PriceInterior,
                         PriceExterior,
                         Surcharges) - PriceReductions
                  <= 95,569.14
```

This computation could also be done dynamically, which would require additional operations in the DSL, e. g., to access the minimum and maximum of the feature domains.

Another problem could involve the weight of a product. For example, in the European Union, a passenger car may not exceed the weight of 3.5 tons. Therefore, to prevent configurations from exceeding this weight, a constraint exists that limits the sum of the weights of all parts accordingly. The weight must always be rounded up to the next kilogram so that the limit is not exceeded unnoticed due to rounding. A test is implemented.

```
require WeightInKG <= 3500
```

6. Conclusion

Automated testing for quality assurance in all areas of computer science has a long history. This ranges from validation and verification in knowledge engineering [17, 18, 19] to a complete portfolio of testing techniques such as unit, integration, system, and acceptance tests in software engineering [8]. The development of tests often requires a similar amount of work as the implementation of the functionality itself. Nevertheless, in the long run this extra effort leads to a reduction of the total work due to the optimization of maintenance, refactoring, and extensibility. The circumstances of an ordered but faulty configuration that has to be recalled from production already justify this effort. Our work has thus presented a novel data structure for domain specific testing in product configuration. It is based on the declarative development of test constraints akin to the development of product knowledge itself. In addition, reasoning modes were introduced to avoid the combinatorial problems associated with simple test cases of expected inputs and outputs and the exponential set of configurations. The options for formulating the test cases must be tailored to the respective developers and their expertise. Our work has exemplified this by presenting a domain specific language to find a compromise between technically aphine and averse developers. Simultaneously, the use of text-based development opens up many opportunities to adapt established software engineering practices.

Nevertheless, several challenges remain. One of them is the evaluation of quality assurance itself using coverage metrics of the tested knowledge. In contrast to software, there is no control flow due to declarativity of our approach, which means that existing metrics such as path or branch coverage are not applicable [20, 21]. Similar to much other work [21, 22], thus more research is needed on how to draw conclusions from our test cases about their quality and the extent of knowledge tested. Another challenge is that the presented framework for testing is itself error-prone, as the formulation of the test constraints is non-trivial. Various papers tackle the task of verifying test suites themselves for this purpose [23, 24, 25]. Finally, the performance of test case execution is a challenge that suffers from the NP complexity of configuration problems. For this, a system to automatically decide which test cases have to be executed depending on the changes made to the knowledge base is conceivable. This is to avoid executing test cases that succeed regardless of occurring regressions and are therefore irrelevant. The execution of only relevant test cases then leads to an improvement of the intended frequent and continuous integration of changes. Ultimately, regression testing represents only one of many tools to ensure knowledge quality such as those used in software engineering. We strive for a portfolio of all these methods in the future and are, at the time of writing, in the process of evaluating the presented methods with industrial partners.

References

[1] A. Felfernig, L. Hotz, C. Bagley, J. Tiihonen, Knowledge-based configuration: From research to business cases, Morgan Kaufmann, Oxford, England, 2014.

[2] A. Falkner, H. Schreiner, Siemens: Configuration and reconfiguration in industry, Knowledge-Based Configuration: From Research to Business Cases (2014) 199–210. doi:10.1016/B978-0-12-415817-7.00016-5.

[3] K. Orsvärna, M. H. Bennick, Tacton: Use of tacton configurator at flsmidth, in: Knowledge-Based Configuration, Morgan Kaufmann, 2014.

[4] I. Nica, F. Wotawa, R. Ochenbauer, C. Schober, H. F. Hofbauer, S. Boltek, Kapsch: Reconfiguration of mobile phone networks, in: Knowledge-Based Configuration, Morgan Kaufmann, 2014.

[5] R. Rabiser, M. Vierhauser, M. Lehofer, P. Günbacher, T. Männistö, Configuring and generating technical documents, in: Knowledge-Based Configuration, Morgan Kaufmann, 2014.

[6] J. Tiihonen, W. Mayer, M. Stumptner, M. Heiskala, Configuring services and processes, in: Knowledge-Based Configuration, Morgan Kaufmann, 2014.

[7] K. Herud, J. Baumeister, O. Sabuncu, T. Schaub, Conflict handling in product configuration using answer set programming, FLoC 2022 ICLP Workshops (2022).

[8] M. Pezzè, M. Young, Software testing and analysis - process, principles and techniques, Wiley, 2007.

[9] J. Tiihonen, M. Heiskala, A. Anderson, T. Soininen, Wecotin - A practical logic-based sales configurator, AI Commun. 26 (2013) 99–131. URL: https://doi.org/10.3233/AIC-2012-0547. doi:10.3233/AIC-2012-0547.

[10] A. A. Falkner, G. Friedrich, A. Haselböck, G. Schenner, H. Schreiner, Twenty-five years of successful application of constraint technologies at siemens, AI Mag. 37 (2016) 67–80. URL: https://doi.org/10.1609/aimag.v37i4.2688. doi:10.1609/aimag.v37i4.2688.

[11] R. Dechter, Constraint processing, Elsevier Morgan Kaufmann, 2003. URL: http://www.elsevier.com/wps/find/bookdescription.agents/678024/description.

[12] S. Jabbour, J. Lonlac, L. Sais, Y. Salhi, Extending modern SAT solvers for models enumeration, in: J. Joshi, E. Bertino, B. Thuraisingham, L. Liu (Eds.), Proceedings of the 15th IEEE International Conference on Information Reuse and Integration, IRI 2014, Redwood City, CA, USA, August 13-15, 2014, IEEE Computer Society, 2014, pp. 803–810. URL: https://doi.org/10.1109/IRI.2014.7051971. doi:10.1109/IRI.2014.7051971.

[13] U. Junker, Configuration, in: F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, volume 2 of *Foundations of Artificial Intelligence*, Elsevier, 2006, pp. 837–873. URL: https://doi.org/10.1016/S1574-6526(06)80028-3. doi:10.1016/S1574-6526(06)80028-3.

[14] J. Baumeister, J. Reutelshoefer, Developing knowledge systems with continuous integration, in: S. N. Lindstaedt, M. Granitzer (Eds.), I-KNOW 2011, 11th International Conference on Knowledge Management and Knowledge Technologies, Graz, Austria, September 7-9, 2011, ACM, 2011, p. 33. URL: https://doi.org/10.1145/2024288.2024328. doi:10.1145/2024288.2024328.

[15] K. Beck, Test Driven Development. By Example, Addison-Wesley Longman, Amsterdam,

2002.

[16] J. Baumeister, J. Reutelshoefer, F. Puppe, Knowwe: a semantic wiki for knowledge engineering, Appl. Intell. 35 (2011) 323–344. URL: https://doi.org/10.1007/s10489-010-0224-5. doi:10.1007/s10489-010-0224-5.

[17] R. Knauf, A. Gonzalez, K. Jantke, Validating rule-based systems: a complete methodology, in: IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.99CH37028), volume 5, 1999, pp. 744–749 vol.5. doi:10.1109/ICSMC.1999.815644.

[18] J. Baumeister, Continuous Knowledge Engineering with Semantic Wikis, habilitation, Universität Würzburg, 2010.

[19] J. Baumeister, Advanced empirical testing, Knowl. Based Syst. 24 (2011) 83–94. URL: https://doi.org/10.1016/j.knosys.2010.07.008. doi:10.1016/j.knosys.2010.07.008.

[20] H. Zhu, P. A. V. Hall, J. H. R. May, Software unit test coverage and adequacy, ACM Comput. Surv. 29 (1997) 366–427. URL: https://doi.org/10.1145/267580.267590. doi:10.1145/267580.267590.

[21] F. Belli, O. Jack, Declarative paradigm of test coverage, Softw. Test. Verification Reliab. 8 (1998) 15–47. URL: https://doi.org/10.1002/(SICI)1099-1689(199803)8:115::AID-STVR146\protect\protect\leavevmode@ifvmode\kern+.2222em\relax3.0.CO;2-D. doi:10.1002/(SICI)1099-1689(199803)8:1\<15::AID-STVR146\>3.0.CO;2-D.

[22] F. Belli, O. Jack, A test coverage notion for logic programming, in: Sixth International Symposium on Software Reliability Engineering, ISSRE 1995, Toulouse, France, October 24-27, 1995, IEEE Computer Society, 1995, pp. 133–142. URL: https://doi.org/10.1109/ISSRE.1995.497651. doi:10.1109/ISSRE.1995.497651.

[23] S. Boroday, A. Petrenko, A. Ulrich, Test suite consistency verification, in: 2008 East-West Design & Test Symposium, EWDTS 2008, Lviv, Ukraine, October 9-12, 2008, IEEE Computer Society, 2008, pp. 235–239. URL: https://doi.org/10.1109/EWDTS.2008.5580145. doi:10.1109/EWDTS.2008.5580145.

[24] P. H. Deussen, S. Tobies, Formal test purposes and the validity of test cases, in: D. A. Peled, M. Y. Vardi (Eds.), Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11-14, 2002, Proceedings, volume 2529 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 114–129. URL: https://doi.org/10.1007/3-540-36135-9_8. doi:10.1007/3-540-36135-9_8.

[25] C. Jard, T. Jéron, P. Morel, Verification of test suites, in: H. Ural, R. L. Probert, G. von Bochmann (Eds.), Testing of Communicating Systems: Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), August 29 - September 1, 2000, Ottawa, Canada, volume 176 of *IFIP Conference Proceedings*, Kluwer, 2000, pp. 3–18.