# Solving Configuration Problems with LogicNG

BooleWorks GmbH


# Solving Configuration Problems with LogicNG

## Abstract

This white paper gives an overview of the open source software library *Logic*$^{NG}$ developed by BooleWorks and its applications to configuration problems.

**LogicNG** LogicNG is a Java library for creating, manipulating and solving Boolean and pseudo-Boolean formulas, containing many algorithms and solver engines for efficiently working with large rule sets arising in product configuration. It is used by many companies around the globe to solve their configuration problems, most notably some German premium car manufacturers. The library is open source under the Apache 2 license and published at GitHub. Extensive documentation is available at www.logicng.org .

**BooleWorks** BooleWorks is a Munich-based company specialized in the application of mathematical logic. The team at BooleWorks is developing algorithms and methods for variant and complexity management. With a strong focus on open source software, “own-your-code”, and transparency, all code developed by BooleWorks is either open source or customer-specific. BooleWorks is a tier 1 software supplier for some of the German premium car manufacturers.

**SofDCar** BooleWorks is a member of the SofDCar project which is part of a research program sponsored by the Federal Ministry for Economic Affairs and Climate Action of Germany (Bundesministerium der Wirtschaft und Klimaschutz, BMWK). All project partners of the SofDCar Alliance are focusing their research on the challenges of future electric/electronic (E/E) & software architecture in vehicles. Within the SofDCar project, BooleWorks is working both on *Logic*$^{NG}$ itself and an extension of its use cases to the software world. The library will be rewritten in Rust, enabling its application within the vehicle and as the base of small, memory-efficient containerized services.

## The Problem

### Mass Customization

When considering interesting use cases for automated reasoning in product configuration, first we have to look at two aspects: product variance, i.e., how many different product configurations are buildable, and product volume, i.e., how many of these configurations are produced every day. Figure 1 illustrates this situation.

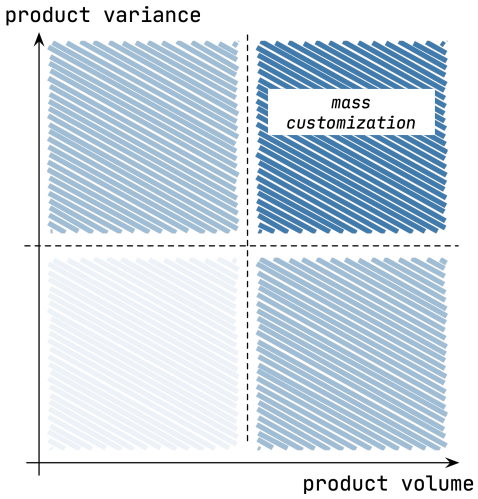


**Figure 1:** Product variance vs. product volume

In today's world there are highly customizable products like airplanes or cruise ships. These products possess high product variance, yet they are not mass-produced (upper left quadrant). The configuration process involves many months of planning and coordination. On the other hand there are products, which are produced in great numbers, yet having only a very limited variance of only dozens or hundreds of different models (lower right quadrant), with a typical example being, e.g., microprocessors. In these cases, all possible configurations can be verified beforehand and there is almost no configuration process during production.

In the upper right quadrant we find products having both a high variance *and* are produced in large numbers. One such class of products are premium cars: A Mercedes E-Series in 2012, e.g., had over $10^{102}$ different buildable configurations and yet thousands of vehicles are produced every day. This is called *Mass Customization*, a term coined by Stan Davis in 1987 and defined by Mitchell M. Tseng and Jianxin Jiao in 1996 as

> producing goods and services to meet individual customer's needs with near mass production efficiency

In mass customization, very interesting but hard configuration problems arise: It is not possible to verify each single buildable configuration by hand or individually, even when this process is automated. Additionally, the mass production aspect allows no room for misconfigurations since a car leaves the moving assembly line every one to two minutes. Line stoppage hence can cost hundreds of thousands of Euros.

Configuration problems do exist in all four quadrants, and *Logic*$^{NG}$ can support the whole process from product development over production to sales here. But the upper right quadrant is where it really shines.

### Common Configuration Problems

Typically, complex products do not have a single set of rules to describe all aspects of the product, but instead there are many different rule sets describing dependencies between features in different domains. A few of these rule sets with examples from the automotive industry are presented in Table 1.

| Domain | Example |
|---|---|
| **High Level Configuration** | |
| **Technical Rules** | the feature for the parking computer depends on the feature for parking sensors |
| **Legal Rules** | in Germany, each vehicle must have a breakdown triangle |
| **Homologation Rules** | in the EU, exhaust emission standard EU6 is mandatory |
| **Marketing Rules** | the US market forces air condition in every vehicle |
| **Bill of Materials (BOM)** | which steering wheel is built in a vehicle is dependent on features like the wheel itself, interior color, multimedia system, … |
| **Electronic Control Unit (ECU) Configuration** | the parameter for the vehicle length of the parking assistant's ECU is dependent on the exterior package and the existence of a trailer hitch |
| **Software Configuration** | different versions of a parking assistant's software require different sensors built in the vehicle and are dependent on the software versions of the camera firmware |

**Table 1:** Different rule sets and examples for an imaginary vehicle

As depicted in Figure 2, these rule sets have intersections, but single rule sets can also describe more variants than other rule sets allow. This is due to the fact that each domain describes only the variance which is relevant to it: a bill of material (BOM), e.g., can describe hardware variants which are currently not sold in any market, but might be in the future.

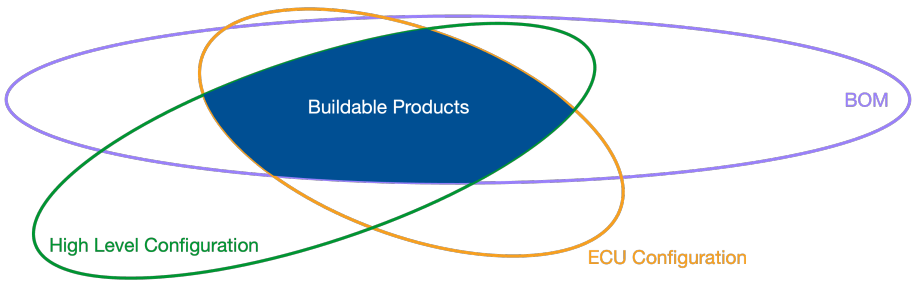


**Figure 2:** Intersection of different rule sets

Each rule set can be verified individually for certain validation criteria, e.g. that there is no contradiction in the set of rules. Further, verifications across different rule sets are common. Usually, the high level configuration (HLC) is used as base rule set, since it describes all buildable products on a certain level (technical, for a certain legislator, in a certain marketing region, ...). Other rule sets are then verified against this HLC to prove certain properties. Table 2 presents some tasks commonly performed and properties commonly validated on the example of a vehicle.

| Domain | Example Tasks and Verifications |
|---|---|
| **General** | • simplification of rules with and without consideration of the high level configuration (HLC)<br>• visualization of rules, transformation of rules into certain formats<br>• evaluation of rules for single configurations |
| **High Level Configuration (HLC)** | • validation of the consistency of the rule set<br>• computation of non-buildable and enforced features<br>• projection of the rule set to certain scopes<br>• computation of impacts of changes in the rule set |
| **Bill of Materials (BOM)** | • verification that positions (e.g. side mirror) are unique, meaning exactly one variant is taken for each buildable vehicle<br>• computation of non-buildable parts in the BOM |
| **Electronic Control Unit (ECU) Configuration** | • verification that parameter values are uniquely defined for each parameter<br>• validation of the consistency between ECUs in the BOM and the ECU configuration<br>• simplification of ECU parameter conditions to speed up flashing on the moving assembly line |
| **Software Configuration** | • solving software update problems<br>• computation and optimization of software update paths<br>• validation of software releases in combination with hardware releases |

**Table 2:** Different rule sets and examples

## The Solutions

### The General Approach

Many of the above mentioned rule sets are formulated as Boolean rules or slight extensions of Boolean algebra. Therefore, algorithms from the area of Boolean logic and automated reasoning are well suited for solving the associated problems. *Logic*$^{NG}$ provides many algorithms to work with Boolean and pseudo-Boolean formulas; the most important ones for solving complex problems being a variety of solvers, namely SAT-, Cardinality-, and MaxSAT Solvers. These can decide the satisfiability of large Boolean formulas, compute configurations, explain conflicts, and optimize solutions for given criteria.

Alternatively, one can compile the problem into a knowledge compilation format like binary decision diagrams (BDD) or disjunctive negation normal forms (DNNF), and then perform computations on this compiled format. This general process is presented in Figure 3.

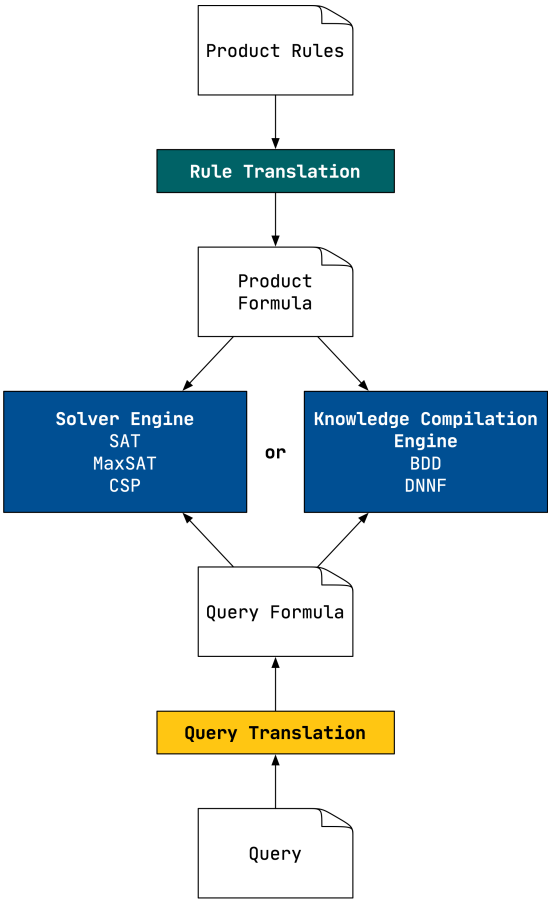


**Figure 3:** The general solving process

The product rules are translated to a Boolean formula, describing the entire solution space. I.e., each solution of this formula is one valid configuration of the product. This product formula is then either loaded onto a solver, or compiled into a knowledge compilation format. These two steps are usually performed only once, and then hundreds, thousands, or sometimes millions of requests are queried against the solver or compiled format.

To do this, the concrete query is also translated into a Boolean formula—the query formula—and then checked against the solver or compiled format. To illustrate this, imagine a vehicle as a product. All product rules are added onto a solver such that each solution found by the solver represents one valid vehicle configuration. Now take as an example the bill of material for this vehicle. For a premium car, typically, there are hundreds of different physical steering wheels, each with its own selection condition specifying in which vehicle the respective steering wheel is used. The question now is, whether there

exists any valid vehicle which does not feature a steering wheel. The translation of this query into a Boolean formula is the conjunction of all possible steering wheels negated. If this query formula is satisfiable together with the product formula on the solver, then there is at least one vehicle which satisfies all product rules, but on the other hand does not satisfy any selection condition of a steering wheel—therefore we found a vehicle without steering wheel. If the product formula and the query formula are not satisfiable together, then we now know that each valid vehicle configuration indeed has a steering wheel.

*Logic*[NG] does not only provide the solvers and knowledge compilers, but also very efficient data structures for formulas, plus many algorithms to generate and manipulate these formulas. This is especially important for implementing product and query translators.

### Manual Testing vs. Automated Reasoning

Traditionally, certain properties of rule sets were verified by manual tests. But with up to $10^{100}$ buildable configurations, each manual testing approach will yield a test coverage of de facto 0%—even when testing billions of different configurations manually. The problem is that if your manual tests do not find an invalid configuration, you never know whether there *is no* invalid configuration or you just *did not find it*. The left graphic in Figure 4 illustrates the situation. The blue space is the set of all buildable variants. The large red area are invalid configurations and the white dots are your tested configurations. In this case your tests do not hit an invalid configuration.

**Figure 4:** Manual testing vs. automated reasoning

However, a product formula as introduced in the last section describes *every* buildable variant, implicitly. When using a solver or knowledge compilation engine we either find an invalid configuration or we get a *mathematical proof* that there can be none. There is no uncertainty like in manual testing. The right graphic in Figure 4 illustrates this: all buildable vehicles are implicitly tested and therefore we quickly find an invalid configuration.

The advantage of always considering the complete solution space translates also to other algorithms. When searching an *optimal product configuration*, e.g., the heaviest configuration, the solution is a mathematical global optimum, not some approximation. When computing *all* possible combinations between a set of features, the result indeed is each mathematically possible configuration, not just a sample of it.

### On-the-fly Computation vs. Compilation

As shown in Figure 3, there are two fundamentally different approaches to answer queries on product rules: (1) using a solver engine, and (2) compiling the rules in a knowledge compilation format. *Logic*$^{NG}$ has support for both approaches. In our real-world projects in the automotive industry, however, we use solver engines > 95% of the time, and knowledge compilation only for special use cases. The big advantage of knowledge compilation is that compiled formats can be very succinct and allow for answering many questions very rapidly (often in constant or linear time). However, the computation time for the compilation itself can grow exponentially. A solver, on the other hand, can take exponential time to answer questions in the worst case, but it allows for very fast initial loading, addition and removal of rules. The following considerations indicate why the solver-approach is frequently superior in practice:

(1) Real industrial problems tend to be easily solvable for modern solver engines—often in a few milliseconds—rendering the performance advantage of knowledge compilations irrelevant.
(2) Large rule sets are very hard to compile into knowledge compilation formats. Typically taking minutes, often it is not even possible at all due to the exponential growth of compilation time.
(3) With some ground-breaking techniques been developed within the last twenty year, SAT solvers have advanced quickly in the recent past. Research on, e.g., BDDs did not progress comparably.

As a practical example, we take a small rule set from a German premium car manufacturer—for large rule sets a compilation into BDDs or DNNFs is infeasible. For this small rule set the compilation into a BDD can take minutes, whereas loading the formulas onto a solver takes only a few hundred milliseconds. Solving such formulas with a modern solver usually takes only few milliseconds—approximately the same time as for a compiled BDD. I.e., in the same time in which the BDD is generated, the solver could have solved hundreds of thousands of queries already. Plus, even when looking at the compiled format, the BDD does not yield any performance advantages. Even worse: in an interactive environment where someone creates or changes rules in the rule set and wants to perform on-the-fly validations and computations, the BDD had to be recompiled after each change.

There are, of course, also useful scenarios for knowledge compilation: usually, when the rule set is small, when not the whole rule set has to be considered, or for visualizing formulas.

Therefore, we want to pursue and support both approaches, the solver-approach and knowledge compilation. To this respect, *Logic*$^{NG}$ on one hand hand provides three different kinds of solver engines:

(1) SAT solvers for solving Boolean and pseudo-Boolean formulas.
(2) Cardinality solvers specialized for formulas with many cardinality constraints.
(3) MaxSAT solvers including partial weighted MaxSAT solvers for optimizations on Boolean and pseudo-Boolean formulas.

On the other hand, the library also features two knowledge compilers:

(1) A BDD implementation with interactive reordering and different variable ordering strategies.
(2) A d-DNNF compiler based on DTrees.

### Decision Problems vs. Optimization Problems

Another interesting question is whether the query at hand is a *decision problem* or an *optimization problem*. A decision problem can be answered with a simple yes/no answer, e.g., if a certain feature is buildable for a given configuration. An optimization problem on the other hand involves attaching weight factors to certain features, e.g., prices or mass in kg, and then searching for an optimal solution by minimizing or maximizing the cumulative weight factors of the configuration. The result would be for example the cheapest product configuration or the heaviest. Whereas decision problems can be solved with both solver engines and knowledge compilation formats, optimization problems usually are solved with optimizing solvers like a MaxSAT solver.

*Logic*$^{NG}$ supports solving with hard and soft rules: a part of the rules can be defined as hard which *must* be fulfilled by every configuration, whereas soft rules *should* be fulfilled. The solver then searches for a solution which satisfies a maximum number of the soft rules. Weighted problems are also possible, where each formula can have a weight and then a solution with a maximum weight is searched for by the solver. These are very powerful tools to model and solve many kinds of real-world optimization problems.

## Real Solutions for Real Problems

### Today's Use Cases

In this chapter we present use cases where *Logic*$^{NG}$ is used today to solve problems in the German automotive industry. All of the presented use cases are implemented in production systems and have been running for many years, some of them used millions of times a day. Many of these use cases are, however, not specific to the automotive industry and could easily be transformed to other application areas.

**High Level Configuration (HLC)** The high level rule set describes every buildable product configuration and therefore often is the base of all subsequent design, product development, production, sales, and after-sales processes. Its correctness is of uttermost importance. The library is used to *validate rule sets* for given criteria, e.g., are there features which are not buildable at any point in time, are there rules which contradict other rules, etc. Our algorithms are used to search for time gaps in rules, e.g., that a new rule supersedes an old one but there is a time gap between the two. The library is also used in interactive rule generation and maintenance processes and checks if a new or modified rule is

consistent with the current rule set, if it can be simplified, or if it contains parts which contradict other rules.

Typically, the high level rule set consists of thousands of rules over many product lines, countries, and releases. Hence, it is often useful to project these rules to arbitrary filter conditions. *Logic*$^{NG}$ can be used to perform these *projections*, or *simplify* the resulting rules, showing which rules are affected by the projection, and also *highlight the differences* within milliseconds.

Another very important supported use case, used in many software systems, is to generate the buildable variance over a certain set of features: a maintainer for breaking systems, e.g., knows exactly which features influence the configuration of the breaking system and wants to know which combinations between these features are buildable with respect to the high level rule set. Very efficient model enumeration algorithms allow to *enumerate these buildable configurations*, even for millions of combinations within seconds.

Also a very popular use case is to *optimize the number of required product configurations*. Thinking of the production of test vehicles, there are hundreds or thousands of requirements for physical test vehicles: For a desert test drive, e.g., someone needs a vehicle with air conditioning, while for a photo shoot a red vehicle in the luxury line is needed, and so on. The question now is: How many physical vehicles do you need to produce *at least* in order to satisfy *all* these constraints. Since test vehicles are very expensive, saving one or two vehicles can be lucrative. Previously, these processes were often performed by hand, taking days or even weeks. With the library's optimization algorithms, the corresponding computations can be performed in minutes, yielding guaranteed minima.

**Ordering Process** *Logic*$^{NG}$ is used in configurators for both in-house product development, and customer touchpoints. Besides the standard functions like supporting an *interactive configuration*, where in each step the remaining selectable features are highlighted, it supports some advanced use cases.

In the case of a conflict, one can easily *explain why a certain configuration is not buildable* by employing one of many available explanation algorithms like, e.g., finding the shortest explanation for the conflict. During the product development process, this can often save many hours of analysing the rule sets manually.

However, if the user explicitly wants to select a feature which contradicts the current configuration, you can *compute possible re-configurations* for user-provided criteria. As such, it is possible to compute the, e.g., cheapest re-configuration or the one with fewest changes to the current order. This is also used in automated processes like validating all currently active orderings every night, computing which vehicles are not buildable any longer because of changes in the rule sets, and proposing possible re-configurations to the user.

**Bill of Materials (BOM)** The bill of material (BOM) maps high level configurations to physical parts of the product. E.g., for a vehicle, a high level feature like “Sport Steering Wheel” gets resolved into many parts like the steering wheel itself but also encompasses cables, mounts, or screws. Its validation is very important since a product-individual 100% BOM is computed for each configuration which is produced. This is called a “BOM explosion”. Errors in the BOM can yield wrong parts at the moving assembly line and, in the worst case, causes line stoppage which potentially costs hundreds of thousands of Euro.

One of the first use cases of *Logic*$^{NG}$ was to *validate the BOM* for correctness criteria, e.g., that each buildable configuration gets exactly one steering wheel. These computations are always performed over the whole product variance, not only for certain vehicles. Therefore errors can be spotted *before* a vehicle is ordered or produced, because *every possible vehicle* is implicitly validated. It is also used, e.g., for computing if there are *parts which are not buildable* in a certain plant for a certain time period. This is key for advanced supply chain management.

Today, our algorithms are also often employed for the traditional *BOM explosion* for a given product. Although this is a standard functionality of every BOM system, the achieved speed is often far superior to these systems. In the automotive context, our algorithms can resolve > 100,000 BOM positions per second on a standard laptop, single-threaded on one processor core.

Thanks to the library’s extensible BDD implementation, it is also used to *visualize BOM positions* and allows a graphical maintenance of parts at a position. It can first visualize the position as a graph, allowing the maintainer to add and remove parts or to draw and re-route edges, and then computes the new resulting constraints for the BOM system.

**Homologation Requirements** With the introduction of the Worldwide harmonized Light vehicles Test Procedure (WLTP) in the automotive industry in 2019, the homologation process and the computation of vehicle-individual $CO_2$ and consumption values got much more complex. *Logic*$^{NG}$ has been used from day one to help in both use cases.

*Computing the best and worst vehicles* with respect to their energy consumption is necessary for the homologation process. Since this is a linear programming problem, a linear optimizer like CPLEX or Gurobi is the best tool for the job. However, the state of the art tools could not yield the performance required to perform tens of thousands of computations a day. Our algorithms have been used to simplify all involved rules and pre-process many of the data in order to improve the performance of the linear optimizers. This helped to speed up the computation time from some minutes to a few seconds.

*Computing the individual $CO_2$ and consumption values* for a vehicle requires the exact weight, aerodynamic values and rolling resistance of that particular vehicle. The library is used to calculate those physical data by efficiently computing the BOM explosion. Since these computations are integrated in the online configurators, speed and response time is extremely important. With our algorithms,

the computation of a single vehicle's individual values requires less than one millisecond, and such operations are being performed millions of times a day in production systems.

**ECU Configuration** Electronic control units (ECU) play an ever growing part in today's vehicles. A modern vehicle has over one hundred ECUs. The validation of both the correct usage of the ECUs as well as their configuration is as important as the validation of the BOM. Since the ECUs of vehicles often are flashed on the moving assembly line with only a short time to flash the right configuration, concise and simplified formulas are required. Last but not least, not only the ECUs are highly configurable, but also the cable harnesses connecting them throughout the vehicle.

Similar to the validation of BOMs, *Logic*$^{NG}$ is used for the *verification of the ECU parameter configuration* for all possible buildable vehicles. In contrast to the BOM, which is usually documented for a vehicle series, ECUs are often documented for all series and product lines. This leads to very large and complex formulas in their configuration. Also some ECUs have thousands of parameters with tens of thousands of different possible parameter values. Once again the speed of our algorithms and solver engines is very important, enabling interactive verification of parameters over all product lines.

The formula simplification algorithms provided in the library are used to *simplify the flash conditions* for ECU parameters, allowing for a fast flash process on the assembly line. Formulas are not only simplified mathematically, but also with additional domain knowledge, providing even shorter representations.

The library is also used to *optimize module building in cable harnesses*. Today's cable harnesses consist of many modules (if they are not customer-specific). But the questions are: how many modules should you design and produce? Are there modules which are always required? Modules which force other modules? Or modules which are not required anymore? By taking take-rates into account one can also optimize the module building process by producing specific modules for popular feature combinations, enabling cost reductions.

## Towards a Software-Defined World

The above use cases have a strong focus on the development, production and sales processes of the hardware of the product. Since the rise of the *Software Defined Vehicle*, software plays an equally important part in all phases of the life cycle of a vehicle.

One of the most important aspects is the *management of software packages*: which versions of a software or library are compatible with which other versions, which packages have dependencies to or restrictions with other packages. This is an area where SAT solvers are traditionally used to resolve package update problems. Linux package managers or the Eclipse plugin manager, e.g., use SAT solvers to resolve their packages. A very small (< 300 lines of codes) proof of con-

cept package solver with different optimization strategies based on *Logic*$^{NG}$ was released here: https://github.com/booleworks/package-solving-poc.

Another use case is to *validate a whole software release* including ECUs, libraries, and software packages. But this is no static process: over-the-air (OTA) updates make it necessary to constantly verify all software packages and updates against regulations like, e.g., the UNECE R 156. A change in a single rule in any of the rule sets (be it high level, BOM, ECU, or software) can for example have an effect on the homologation of a vehicle. A software update to a motor ECU, e.g., could change its emission values and invalidate a homologation. Within the SofDCar project, BooleWorks is working with its partners from industry and research to develop holistic data models for all relevant data and rule sets, which can then be validated both for all planned but also all already produced vehicles, perpetually and automatically.

A use case which is already present today, but which will play an even more important role in the future is the whole aspect of *after-sales software and service add-ons*. While the customer today can buy some new services online, in the future, whole vehicle features will be available on-demand. Thus, the *computation of selectable services and software features* will play an important role, as well as *re-configuration algorithms*, enabling customers to buy a feature which is currently not possible in their vehicle.

## Further References

BooleWorks' employees published many scientific articles on algorithms for solving configuration problems over the last years. Comprehensive introductions are provided by two PhD theses:

- New Formal Methods for Automotive Configuration (2014) by Christoph Zengler
- SAT-based Analysis,(Re-) Configuration & Optimization in the Context of Automotive Product Documentation (2018) by Rouven Walter

## Contact

**BooleWorks GmbH**
Radlkoferstr. 2
81373 Munich, Germany

Mail: info@booleworks.com
Web: https://www.booleworks.com

LogicNG: https://www.logicng.org

The SofDCar project is funded by Federal Ministry for Economic Affairs and Climate Action