Thesis for the degree of Doctor of Philosophy

On Formal Methods for Large-Scale Product Configuration

Alexey Voronov

Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2013

On Formal Methods for Large-Scale Product Configuration
Alexey Voronov
ISBN 978-91-7385-790-1

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3471
ISSN 0346-718X

Department of Signals and Systems
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Telephone + 46 (0) 31 – 772 1000

Cover: Supervisor for interactive product configuration, see Figure 5.8 on page 63.

Typeset by the author using LaTeX.

Printed by Chalmers Reproservice
Göteborg, Sweden 2013

to my family

Abstract

In product development companies mass customization is widely used to achieve better customer satisfaction while keeping costs down. To efficiently implement mass customization, product platforms are often used. A product platform allows building a wide range of products from a set of predefined components. The process of matching these components to customers' needs is called product configuration. Not all components can be combined with each other due to restrictions of various kinds, for example, geometrical, marketing and legal reasons. Product design engineers develop configuration constraints to describe such restrictions. The number of constraints and the complexity of the relations between them are immense for complex product like a vehicle. Thus, it is both error-prone and time consuming to analyze, author and verify the constraints manually. Software tools based on formal methods can help engineers to avoid making errors when working with configuration constraints, thus design a correct product faster.

This thesis introduces a number of formal methods to help engineers maintain, verify and analyze product configuration constraints. These methods provide automatic verification of constraints and computational support for analyzing and refactoring constraints. The methods also allow verifying the correctness of one specific type of constraints, item usage rules, for sets of mutually-exclusive required items, and automatic verification of equivalence of different formulations of the constraints. The thesis also introduces three methods for efficient enumeration of valid partial configurations, with benchmarking of the methods on an industrial dataset.

Handling large-scale industrial product configuration problems demands high efficiency from the software methods. This thesis investigates a number of search-based and knowledge-compilation-based methods for working with large product configuration instances, including Boolean satisfiability solvers, binary decision diagrams and decomposable negation normal form. This thesis also proposes a novel method based on supervisory control theory for efficient reasoning about product configuration data. The methods were implemented in a tool, to investigate the applicability of the methods for handling large product configuration problems. It was found that search-based Boolean satisfiability solvers with incremental capabilities are well suited for industrial configuration problems.

The methods proposed in this thesis exhibit good performance on practical configuration problems, and have a potential to be implemented in industry to support product design engineers in creating and maintaining configuration constraints, and speed up the development of product platforms and new products.

Keywords: Product configuration, constraint satisfaction, Boolean satisfiability, knowledge compilation, supervisory control theory.

Acknowledgments

I would like to thank everyone without whom this thesis would not be possible. First of all, my main supervisor, Knut Åkesson, thank you for your guidance, enthusiasm, energy and ideas. My second supervisor, Martin Fabian, thank you for discussions, good feedback and an always welcoming open door. Bengt Lennartson, thanks for your encouragements during the project, and for excellent management of the research group. All members of our automation group, thanks for all the discussions, DK-meetings, kick-off meetings, fika-times, floorball matches and afterworks.

Thanks to all the people who made day-to-day work run smoothly, especially to Madeleine, Agneta, Christine, Natasha, Lars and Ingemar.

Thanks to all external co-authors with whom I had a pleasure to work: Koen Claessen, Mary Sheeran, Niklas Sörensson, Niklas Een, Anna Tidstam, Johan Malmqvist and Fredrik Ekstedt. I would also like to thank all academic and industrial members of the Wingquist project on Configuration Rule Management.

I would like to thank Anna Reymer, Knut Åkesson and Martin Fabian for reading this thesis and giving valuable comments. Without you, this thesis would be nowhere near as good.

Special thanks to Jörgen Sjöberg, without whom I would never come to Sweden in the first place.

My family. Thanks for believing in me. Mom, sister, grandma, many others. Anna, you bring joy to my life. Thank you so much!

Alexey Voronov
Göteborg, December 2012

This work was carried out within the Wingquist Laboratory VINN Excellence Centre at Chalmers University of Technology. It was supported by the Swedish Governmental Agency for Innovation Systems (Vinnova). The work was also supported by Swedish Foundation for Strategic Research through the ProViking program.

List of Publications

This thesis is based on the following appended papers:

Paper 1. Alexey Voronov, Knut Åkesson, Anna Tidstam, Johan Malmqvist and Martin Fabian. *Toward better support for authoring and maintaining product configuration constraints.* Submitted (2012).

Paper 2. Alexey Voronov, Knut Åkesson, Anna Tidstam and Johan Malmqvist. *Verification of Item Usage Rules in Product Configuration.* Proceedings of 9th International Conference on Product Lifecycle Management PLM-12, Montreal, Canada, 2012.

Paper 3. Alexey Voronov, Knut Åkesson and Fredrik Ekstedt. *Enumerating partial configurations.* Proceedings of Configuration Workshop at 22nd International Joint Conference on Artificial Intelligence IJCAI-11, Barcelona, Spain, 2011.

Paper 4. Koen Claessen, Niklas Een, Mary Sheeran, Niklas Sörensson, Alexey Voronov and Knut Åkesson. *SAT-Solving in Practice, with a Tutorial Example from Supervisory Control.* Journal of Discrete Event Dynamic Systems 19(4), pp. 495–524, 2009.

Other relevant publications co-authored by Alexey Voronov:

Anna Tidstam, Lars-Ola Bligård, Fredrik Ekstedt, **Alexey Voronov**, Knut Åkesson, Johan Malmqvist. *Development of Industrial Visualization Tools for Validation of Vehicle Configuration Rules.* Proceedings of 9th International Symposium on Tools and Methods of Competitive Engineering, pp. 14, 2012.

Sajed Miremadi, **Alexey Voronov**. *Symbolic Reduction of Guards in Supervisory Control Using Genetic Algorithms.* Technical report. Göteborg: Chalmers University of Technology, 2012.

Alexey Voronov, Knut Åkesson. *Verification of Process Operations Using Model Checking.* Proceedings of IEEE International Conference on Automation Science and Engineering CASE'2009. pp. 415-420, 2009.

Alexey Voronov, Knut Åkesson. *Verification of Supervisory Control Properties of Finite Automata Extended with Variables.* Technical report. Göteborg: Chalmers University of Technology, 2009.

Alexey Voronov, Knut Åkesson. *Supervisory Control using Satisfiability Solvers.* Proceedings of 9th International Workshop on Discrete Event Systems WODES'2008, pp. 81-86, 2008.

List of Acronyms

AI	–	Artificial Intelligence
API	–	Application Programming Interface
BDD	–	Binary Decision Diagram
BMC	–	Bounded Model Checking
BOM	–	Bill of Materials
CNF	–	Conjunctive Normal Form
CSP	–	Constraint Satisfaction Problem
CTL	–	Computation Tree Logic
DNNF	–	Decomposable Negation Normal Form
sd-DNNF	–	Smooth Deterministic DNNF
EFA	–	Extended Finite Automaton
FPT	–	Fixed Parameter Tractability
FSA	–	Finite State Automaton
IUR	–	Item Usage Rule
LTL	–	Linear Temporal Logic
MDD	–	Multivalued Decision Diagram
MUS	–	Minimal Unsatisfiable Subformula
PDM	–	Product Data Management
PLM	–	Product Lifecycle Management
SAT	–	Boolean Satisfiability Problem
SCT	–	Supervisory Control Theory
SMI	–	Set of Mutually-Exclusive Required Items

Contents

Part I

Introductory chapters

Chapter 1

Introduction

Modern manufacturing is very challenging. On one hand, broad competition pushes manufacturers to keep costs down. One of the most widespread practices for keeping costs down is mass production, pioneered by Henry Ford more than a century ago. By extreme standardization and unification it is possible to reduce costs. A famous quote by Ford (1922) says: “Any customer can have a car painted any colour that he wants so long as it is black”, which emphasizes standardization. On the other hand, customers want individualized solutions. For example, a buyer of a commercial truck knows how the vehicle will be used, and does not want to pay for extra cargo capacity or driving range. Such individualization contradicts mass production.

Mass customization, envisioned by S. M. Davis (1987), bridges the gap between mass production and custom-made products. Mass customization is a production strategy focused on the broad provision of personalized products and services produced almost as cheaply as mass products (Pine II et al. 1993; Piller and Stotko 2002; Hvam et al. 2008; Fogliatto et al. 2012). The key to implementing mass customization is to use *product families* and *product platforms*. A product family is a group of related products that is derived from a product platform to satisfy a variety of market niches (Simpson et al. 2006). A product platform is a “set of common components, modules, or parts from which a stream of derivative products can be efficiently developed and launched” (Meyer and Lehnerd 1997). The process where customer needs are matched to the company’s standardized components and procedures is called *configuration*. Configuration, according to Mittal and Frayman (1989), is “a special type of design activity, with the key feature that the artifact being designed is assembled from a set of pre-defined components that can only be connected together in certain ways”. Given the complexity of most modern products, configuration would be almost impossible without information system support.

There are different approaches to implement information systems for configuration. Sabin and Weigel (1998) divide them in three groups: *case-based*, *rule-based* and *model-based*. Case-based systems (Kolodner 1992) store all previously sold products as cases, and use these cases as a basis for each new order, possibly making necessary design adaptations either manually or automatically. The advantage of case-based systems is that they do not require large amounts of work upfront for designing a product platform. However, the absence of a carefully designed platform prevents the manufacturer from fully realizing the benefits of unification and economy of scale. Rule-based systems (Hayes-Roth 1985) use *production rules* of the form IF *condition* THEN *consequence* to encode what actions should be performed to obtain a valid configuration, and when each action should occur with respect to other actions. Rule-based systems suffer from severe maintenance problems (Barker et al. 1989) due to the tight coupling between the domain knowledge and the inference engine, both encoded in the same set of rules. Model-based systems were developed to address the limitations of case-based and rule-based systems. The main assumption of model-based system is the existence of a *model* of the product being configured. Such a model consists of decomposable entities and interactions between their elements. The model facilitates the separation between what is known and how the knowledge is used (Hamscher 1992).

Model-based systems can be further sub-classified. The models for model-based configuration can be created using *description logic* (McGuinness and Wright 1998; McGuinness 2003), *features* (Kang et al. 1990; Thiel and Hein 2002; Batory 2005), *ontologies* (Asikainen et al. 2007; Yang et al. 2008), *answer set programming* (Soininen et al. 2001), *preference programming* (Junker and Mailharro 2003) and *constraints* (Mittal and Frayman 1989; Junker 2006). This thesis focuses on constraint-based systems. Constraints provide a simple yet flexible and powerful framework for modeling rapidly-changing products (Schuh et al. 2009). Constraint-based systems are widely used in automotive manufacturing companies like Renault (Amilhastre et al. 2002; Astesana, Cosserat, et al. 2010), DaimlerChrysler AG (Sinz et al. 2003) and Volvo Trucks (Lindroth 2011).

High complexity of constraints, as well as frequent introduction of new products and components, makes manual handling of configuration constraints error-prone and time consuming. Software tools can help engineers analyze and maintain configuration constraints. A number of methods and tools have been reported in the literature that aim to help engineers, including tools for verification and validation of knowledge-based systems (Gupta 1993; Preece et al. 1997; Tsai et al. 1999; Desharnais et al. 2011), verification of automotive configuration data (Amilhastre et al. 2002; Sinz et al. 2003; Astesana, Bossu, et al. 2010; Astesana, Cosserat, et al. 2010), visualization tools for dealing with decisions that cannot yet be automated (Baumeister and Freiberg 2010), virtual builds (Fuxin 2005), automatic analysis of feature models (Benavides et al. 2010), and refactoring of feature models (Alves et al. 2006; Thüm et al. 2009). However, being specialized, these methods and systems do not cover all possible configuration problems. For example, to the best of authors knowledge, the problem of efficient enumeration of valid partial configuration has not been addressed, and the problem of supporting engineers in analyzing constraints and possibly improving them was not covered exhaustively.

This thesis focuses on the following research questions:

1. What kind of computer support can be implemented to help engineers maintain, verify and analyze product configuration constraints? (Approached in Papers 1, 2 and 3).

2. How to enumerate valid partial configurations efficiently? (Paper 3).

3. How to compactly represent product configuration data for answering product configuration questions efficiently? (Approached in Chapters 4 and 5).

The amount and complexity of constraints make analysis computationally demanding: there might be tens of thousands of constraints and 10^{100} of possible products. Moreover, configuration problem can often be seen as a generalization of the well-known problem of Boolean satisfiability[1], which is a classical problem that belongs to the set of NP-complete problems (Cook 1971), and to date there is no algorithm known that can solve an arbitrary problem instance with a time complexity that is better than exponential in the size of the input (Hertli et al. 2011). However, there is a lot of work done on handling practical instances of NP-complete problems, for example, in the hardware verification community (Burch et al. 1990), which resulted in efficient methods to solve generic satisfiability problems. These methods include, for example, Binary Decision Diagrams (Bryant 1986), Boolean Satisfiability Solvers (Biere, Heule, et al. 2009) and Constraint Programming (Apt 2003; Dechter 2003). This thesis identifies important challenges in working with configuration constraints that can be tackled using recent advancements in methods and tools for solving satisfiability problems.

The main contributions of this thesis are:

1. A number of methods for automatic verification of configuration constraints and for computational support of manual inspection of constraints (Paper 1).

2. A method for verifying the correctness of one specific type of constraints (Item Usage Rules) for sets of mutually-exclusive required items, and a method for automatic verification of equivalence of different formulations of the constraints (Paper 2).

3. Three methods for enumerating valid partial configurations efficiently, and benchmarking of the methods on an industrial dataset (Paper 3).

4. A novel encoding of configuration data that allows checking the validity of partial configurations without exhaustive search, by using Supervisory Control Theory introduced by Ramadge and Wonham (1989); the encoding is suitable for configuration tools that are interactive (Chapter 5).

5. A method for solving Supervisory Control Theory problems—namely synthesis of deadlock-free and controllable supervisors—using Boolean satisfiability solvers (Paper 4).

[1] It is difficult to give a precise reference for Boolean satisfiability problem: logic as a science dates back to Aristotle (384-322 B.C.) (Johansen and Rosenmeier 1998); Boolean functions and variables are named after George Boole (1815-1864) who laid the foundations for an algebraic notation for logic, and this notation was later popularized by William Stanley Jevons (1835-1882) (Gardner 1958); the most widely-cited algorithm that initiated active research in computer methods for solving Boolean satisfiability problem is due to M. Davis and Putnam (1960). More historical notes can be found in the book "Logic machines and diagrams" by Gardner (1958), and more details about the state-of-the-art in Boolean satisfiability can be found in "Handbook of Satisfiability" edited by Biere, Heule, et al. (2009)

6. A prototype implementation of a product configuration engine.

The methods presented in this thesis were implemented in a prototype for rapid experimentation with configuration problems. Methods for enumerating valid partial configurations (Paper 3) and for explaining invalid partial configurations (Paper 1) were further developed for an automotive company by a spin-off company from this research project, Confirmlogic AB, and a pilot project for integrating the methods into daily work process was initiated at that automotive company.

This thesis consists of two parts. Part I is a general introduction to the field and puts the appended papers into context, except for Chapter 5, which has not been published before and is a novel contribution of this thesis. Part II contains the appended papers.

Chapter 2

Challenges in working with configuration constraints

Creation and maintenance of configuration constraints is an important part of development of mass-customized products. The constraints have to be developed before the sales process can start, to specify what can and can not be produced. While ordering the product, a customer can customize or configure the product within the configuration constraints. Once a customer orders a product, the assembly process typically starts. This workflow is illustrated in Figure 2.1.

The process of developing configuration constraints consists of three steps, as illustrated in Figure 2.1: authoring, verification and validation, and release (Watts 2012; *VDA 4965* 2010). Among these tree steps, the main contributions of this thesis are in the second step (verification and validation), but some contributions affect also the authoring step. The third step (release) is where configuration constraints are made available for use by sales and manufacturing departments. The following sections introduce the authoring and the verification and validation steps.

2.1 Authoring

Authoring of configuration constraints can be seen as a knowledge acquisition activity of Knowledge Based Systems. According to Neubert (1993), authoring consists of four standard steps: *elicitation, interpretation, formalization,* and *implementation.* A request to modify a product initiates the *elicitation* step, where a product design engineer, acting as a domain expert, creates a natural language description of what should be done. The natural language description is then translated into a semi-formal description in the *interpretation* step. The interpretation step is needed between the elicitation and formalization steps, as a mediator between knowledge (or information technology) engineers and domain experts (represented by product design engineers) (Angele et al. 1998). In the *formalization* step, a formal logic description of the constraints is created from the semi-formal description. In the last step, *implementation*, the constraints are stored in an information system, for example, by typing the constraints into an editor.

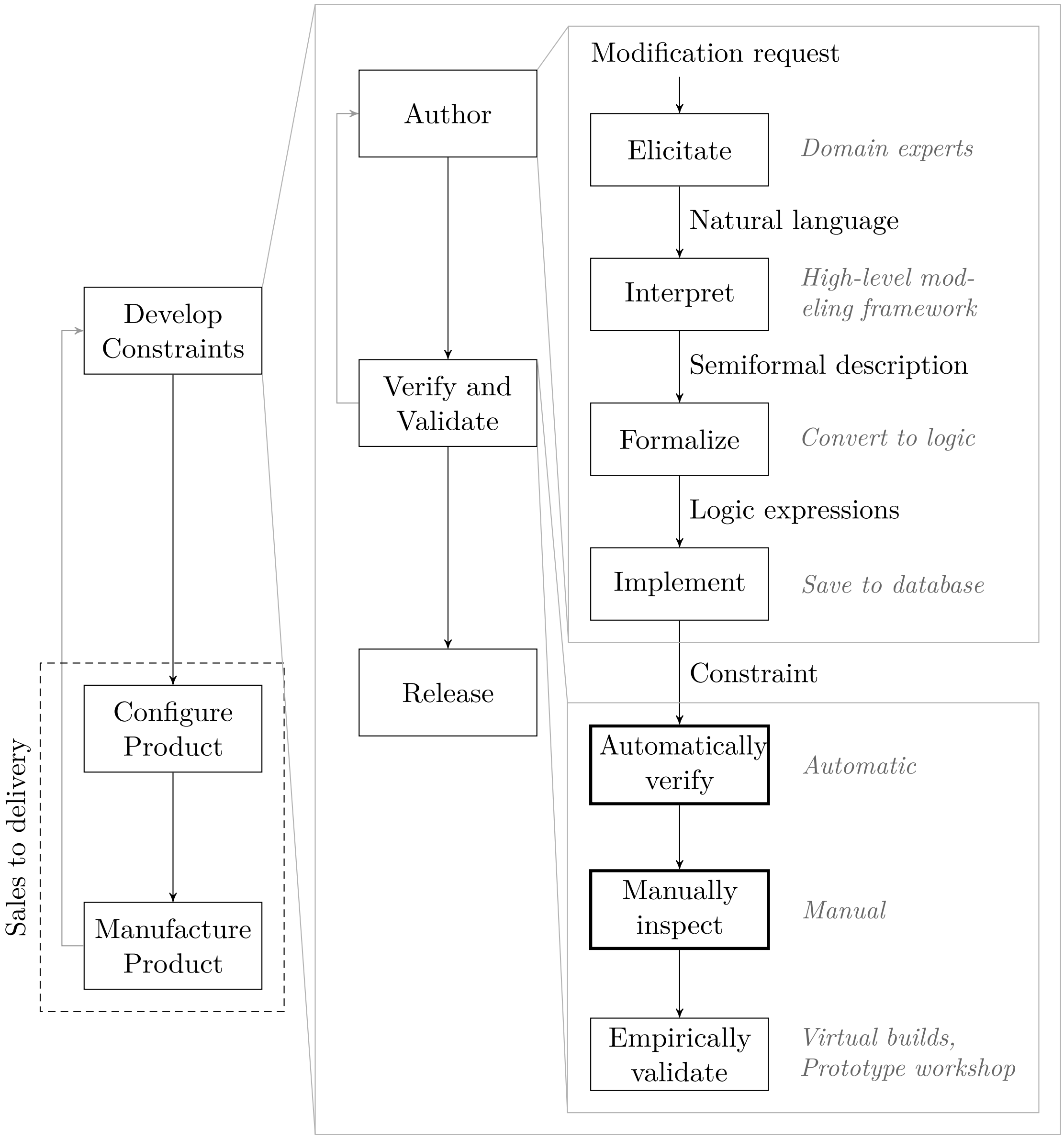


Figure 2.1: Configuration constraints workflow.

Product design engineers author two types of constraints, according to the two tiers of configuration introduced by Haag (1998): high-level customer oriented configuration constraints and low-level manufacturing oriented configuration constraints. The high-level constraints are eventually released for use by the sales department, while the low-level constraints that specify which parts to use for which customer order—together with drawings, assembly instructions and other necessary information—are released to the manufacturing department. This process is illustrated in Figure 2.2.

In the high-level customer-oriented configuration, products are broken down into configurable parts and features referred to as *families*. In a valid product, each family assumes exactly one *variant* from a predefined finite set. Later we give a formal definition to these terms. Combinations of variants will from here on be called *configurations*. A *complete* configuration has exactly one variant assigned to each family in the complete set of families. A *partial* configuration has variants assigned to some families, but not to all. Some configurations are not technically buildable, and some are not desirable for marketing or legal reasons. Configurations that fulfill all limitations are called *valid*. A partial configuration is *valid* if it can be extended to a valid complete configuration. *Variant constraints* define which configurations are valid and which are invalid.

An example of how high-level configuration constraints might look is shown in Table 2.1. The constraints appear in several stages. The first stage is *Authorization*, its constraints specify which variants are allowed for each product type, where a *product type* is a coarse partition of products into types. For example, for an automotive manufacturer such types could be sedan “S60” or station wagon “V70”. The second stage, *Irrelevant configurations*, are constraints that come from marketing, planning and legal departments, which specify configurations that are not to be sold, thus should not be designed. Such irrelevant configurations are depicted by the connected arrows in Table 2.1. For example, the 1.2L engine should not be combined with the *Sport* model. The third stage is where engineers add constraints that forbid some configurations due to engineering reasons, for example, physical constraints or safety, such as the *Sport* model is not to be manufactured with *Diesel* engine. Partial configurations forbidden by engineers are also depicted by connected arrows in Table 2.1.

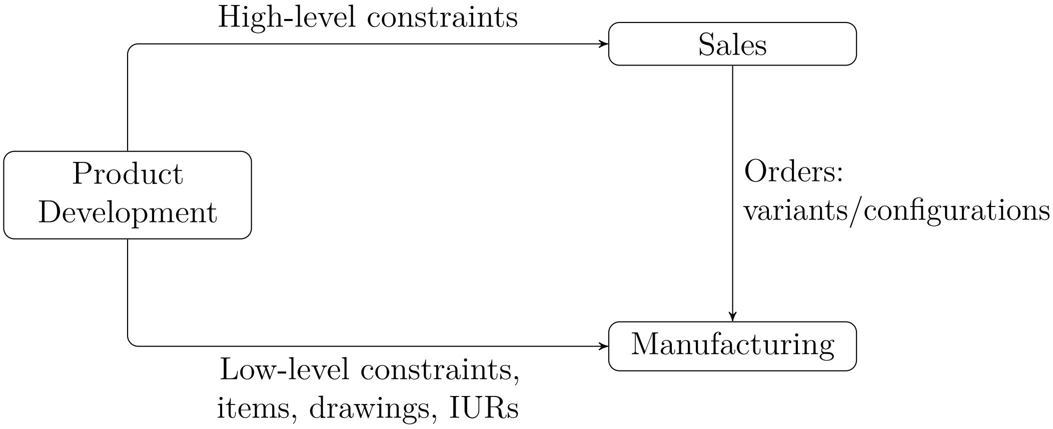
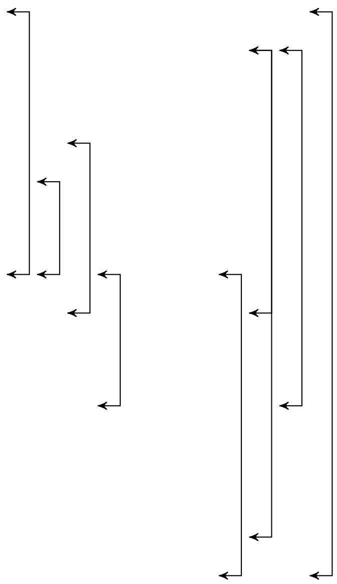

High-level constraints
Sales
Product Development
Orders: variants/configurations
Manufacturing
Low-level constraints, items, drawings, IURs

Figure 2.2: Two-tier configuration.

Table 2.1: Example of possible configuration constraints. *Authorization* constraints specify product type to variant relation. *Irrelevant configurations* constraints specify which configurations are forbidden by, for example, marketing, planning and legal departments. *Engineering restrictions* constraints specify which configurations are forbidden due to engineering reasons.

Families and Variants	Authorizations by Product Type			Irrelevant configurations	Engineering restrictions
	Type A	Type B	Type C		
Volume					
1.2	✓	✓	✓		
1.6	✓		✓		
Turbo					
Yes	✓	✓	✓		
No	✓	✓	✓		
Sport					
Yes	✓	✓	✓		
No	✓		✓		
City					
Yes	✓	✓	✓		
No	✓		✓		
Fuel					
Gasoline	✓	✓	✓		
Diesel	✓	✓	✓		



In the low-level manufacturing-oriented configuration, the structure is defined by *items*. Each item can be either included or not in a *Bill of Materials* (BOM). *Item Usage Rules* (IURs) define which items are selected for each concrete product based on the customer selection of variants for families. IURs, in the form presented here, were reported to be used in automotive industry in (Tidstam and Malmqvist 2010). An IUR is a production rule of the form IF *condition* THEN *item*, where *condition* is a complete or partial configuration (from the high-level customer-oriented tier) that triggers the inclusion of the *item* into the BOM. IURs can be considered as a method for encoding part lists; for a comparison of a number of methods of encoding part lists see, for example, (Sinz 2006). An example of IURs is shown in Table 2.2, where each line contains one IUR. The left part of the table (*Families and variants*) contains the *inclusion condition* part of the IUR, and the right part contains the item to be included in a BOM. For example, the first row of Table 2.2 says that IF the customer ordered family *Volume* to take variant *1.6*, *Turbo* and *Sport* families to take variants *Yes*, and family *City* to take variant *No*, THEN the 1.6L turbocharged engine item denoted *E16T* should be included into the BOM for assembly. An item, such as *E12* in Table 2.2, can be included by several IURs, and identical inclusion conditions can participate in different IURs, thus forcing inclusion of several items.

Table 2.2: IURs. In each row a condition for including an item is specified. For example, the first row specifies that if Volume=1.6 and Turbo=yes and Sport=yes and City=no and Fuel=gasoline then and only then item E16T should be included. For multi-row items (*E12* in this example), an item is included if and only if any of the rows is satisfied.

Families and variants					
Volume	Turbo	Sport	City	Fuel	*Item(s)*
1.6	yes	yes	no	gasoline	*E16T*
1.6	no	no	no	diesel	*E16D*
1.2	no	no	yes	gasoline	*E12*
1.2	no	no	no	gasoline	*E12*

All configurations that are valid with respect to high-level configuration constraints have to be fully designed on the low-level, including specifications of items, IURs and detailed items designs, to be ready for a customer order. This is a so-called *assemble-to-order* strategy, which requires all designs for products that customers can order to be done beforehand, as opposed to an *engineer-to-order* strategy, where a design is created only when a customer orders it. Assemble-to-order allows shorter delivery times compared to engineer-to-order, and is thus the strategy of choice in most automotive companies. However, in the truck industry some companies combine assemble-to-order with more individual solutions that may require additional engineering support.

There can still be a large number of allowed configurations even after authorizations and other constraints have removed many configurations. For instance, there are about 10^{21} possible "Renault Trafic" vehicles (small delivery vans) at Renault (Astesana, Cosserat, et al. 2010), and at least 10^{103} car configurations possible to order for the E-class line of Mercedes-Benz (Kübler et al. 2010). Such a large number of complete vehicle designs is impossible to create explicitly. Instead, the product is broken down into loosely coupled *function groups*, and the complete design of a product is a set of designs from function groups. If we take a hypothetical example of breaking down a product into 20 function groups, and if each function group can be represented by one design chosen from 10, then these $20 \times 10 = 200$ (sub)designs will describe 10^{20} complete products. To implement such product breakdown, each function group is connected with a subset of families. Engineers in a function group have to create detailed designs for all valid partial configurations only within the subset of families. To conform to the assemble-to-order strategy, an engineer responsible for a function group has to prepare all design documents beforehand for all possible valid partial configurations within the function group. If some partial configurations cannot possibly result in a valid product, for example, due to physical limitations, an engineer has to create constraints to prevent customers from ordering such configurations.

Engineers rarely create the whole product platform from scratch. Instead, they do incremental additions to the product offering. Consider, for example, introducing a car with an electrical engine, or even with a hybrid powertrain. To introduce a new engine, it is necessary to add a new variant to the families responsible for the engine. Most likely, the new engine will not work for all configurations, thus it is necessary to add some variant rules to specify when the new engine can be selected. It is also necessary to create IURs that will specify concrete items to be used in the assembly whenever a customer selects a configuration with the new variant.

There might be several problems with the new constraints and IURs, for example, they might forbid configurations that otherwise should be valid, or they can allow something that will not be possible to assemble, or the new rules might simply be redundant and unnecessary. Some of these problems can be discovered as late as on the manufacturing floor, which might result in costly manual adjustments or re-negotiations of the order with the customer. To prevent proliferation of errors from development to manufacturing, the configuration data should be fully verified and validated before it is released to the sales and manufacturing departments.

2.2 Verification and validation

Once the constraints have been authored, the next step in configuration constraints development is verification and validation. According to Boehm (1984), verification is ensuring that the system is built right, while validation is ensuring that the right system is built. Meseguer and Preece (1995) clustered verification and validation activities into four groups: inspection, static verification, empirical testing and empirical evaluation. *Inspection* is performed manually by domain experts—"by eye"—to detect semantically incorrect knowledge in the knowledge base. By manual inspection we will also mean activities involving computational methods that nevertheless require domain expert knowledge to make a decision about the correctness of the system. *Static verification* checks the consistency of a knowledge base using computational support. *Empirical testing* checks correctness by executing the system on sample data sets. Static verification and empirical testing are both combined in this thesis into a group *automatic computation*, since both have potential to be performed automatically, that is, by a software tool, algorithmically, without user intervention. *Empirical evaluation* checks suitability of the configuration constraints for the final user, including manufacturing and sales departments. Prototype workshops and virtual builds, among other methods, can be used to perform such validation (Fuxin 2005).

The verification and validation stage of the constraints development process has gained significant attention in the scientific community, not the least for its high complexity but also due to its wide application to all knowledge-based systems. Over the years a number of tools and methods for verification and validation of knowledge based systems have been created, see e.g. (Preece et al. 1997; Tsai et al. 1999) for reviews. These tools include *consistency checkers*, which detect conflicting and redundant knowledge in a knowledge base; *completeness checkers*, which detect missing or deficient knowledge; *testing tools*, which check correctness using test cases. Despite great attention, however, these tools do not cover specific needs of design engineers in the automotive industry. Feature models (Kang et al. 1990) were proposed for use in automotive product configuration (Thiel and Hein 2002), but have not found widespread use there, possibly due to high duplication of variants and difficulties with introduction of changes, as discussed in (Bühne et al. 2004). However, feature models enjoyed a substantial amount of attention, for example as a tool for modeling software, and as such, a number of methods have been developed for automated analysis and verification. Benavides et al. (2010) reviewed 53 studies and presented a catalog with 30 operations for automated analysis of feature models. Due to a connection between feature models and constraints (Batory 2005), some of the ideas can be readily applied to support product design engineers in automotive companies. Sinz and Küchlin were the first to address the needs of product development engineers in automotive companies by introducing formal methods for verification of product configuration data (Küchlin and Sinz 2000; Sinz et al. 2003), including checks for inadmissible variants, superfluous items, inclusion of mutually-exclusive items, necessary variants and temporal consistency. Later, a number of verification questions, including consistency, validity, completeness, conflict analysis and model counting, were proposed by Astesana et al. (Astesana, Bossu, et al. 2010; Astesana, Cosserat, et al. 2010) for working with product configuration at Renault. Astesana and co-workers also proposed the development of new solvers to handle such questions, without considering how to compute the answers to these questions with the existing solvers. However, there are still problems that manufacturing companies are facing that have not been addressed in the literature. The remainder of this chapter addresses such problems.

2.2.1 Automatic verification

Automatic verification can catch errors without the need for human intervention, and may be executed every time the new constraints are introduced, or old ones changed. This subsection introduces three problems that can be addressed by automatic verification. Originally, these problems were introduced in Papers 1 and 2.

Verifying that new rules do not forbid reference configurations

The complexity and interplay between configuration constraints make it difficult to figure out whether a new constraint is "correct" or not. For example, adding a constraint that forbids configurations that should normally be valid is clearly undesirable. To help prevent situations like these, Paper 1 proposes to create a kind of a "safety net" around the rules, using a number of *reference configurations*. These configurations must always be possible to build, and if they become invalid due to some rules, then more thorough analysis is required.

The reference configurations can be either *complete*, involving variants for all families, or *partial*, with variants assigned only to a subset of families. Partial reference configurations are valid only if they can be extended to valid complete reference configurations. Verifying that a complete reference configuration satisfies a new constraint is easy, while it is much more difficult to verify whether a partial reference configuration satisfies the constraint.

Reference configurations, as well as their counter-part *forbidden reference configurations* that must always remain invalid, are illustrated in Figure 2.3. Reference configurations can also be used as positive and negative examples for model-based diagnosis (Felfernig, Friedrich, Jannach, et al. 2004).

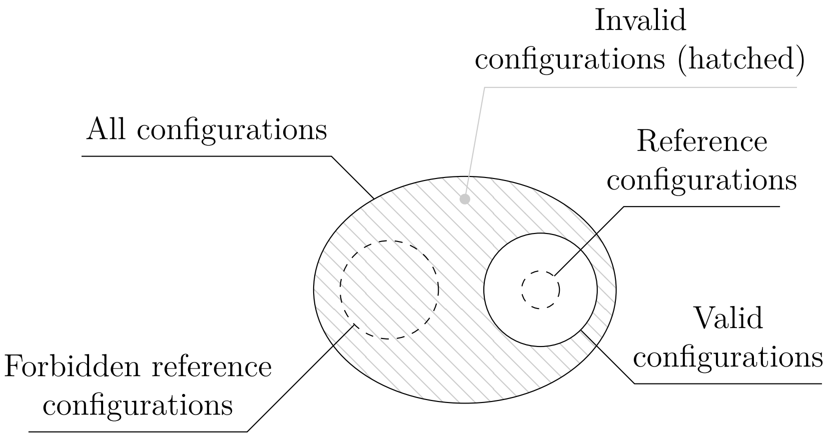
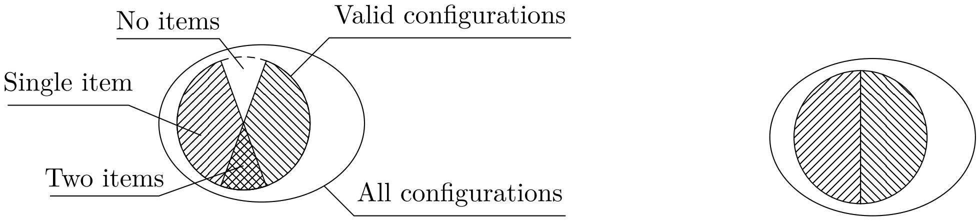


Figure 2.3: Configurations space.

Verifying Item Usage Rules for Mutually-Exclusive Items

IURs specify the connections between variants and items. Since an IUR specifies how an item depends on families' variants, but not on other items, it is easy to end up in a situation where, for example, a car has no engine. Such *at-least-one* condition for a car must be satisfied by, for example, steering wheel items, chassis items, cabin items, windshield items etc. Many of these examples also have a corresponding *at-most-one* condition, for example, a car must have only one steering wheel (there are exceptions though: hybrid cars have more than one engine, some waste collection trucks have two steering wheels etc). Together, at-least-one and at-most-one conditions form *exactly-one* conditions. A set of items that must satisfy an exactly-one condition will be called *Set of Mutually-exclusive required Items* (SMI) (Tidstam, Bligård, et al. 2012; Voronov, Åkesson, Tidstam, et al. 2012), where *mutually-exclusive* means that no two items are allowed together, and *required* means that at least one item is necessary (SMIs are also called *generic items* (Veen 1992)). Exactly-one conditions can be illustrated as in Figure 2.4, which highlights that every valid configuration should have exactly one item assigned from a SMI.

Since there is no support for defining relations directly between items, and since it is necessary to maintain backward compatibility with the existing system, Paper 2



(a) Incorrect: some configurations with one item, some with two, and some with none.

(b) Correct: disjoint and covers all configurations (ensures both *at most* and *at least* one item per configuration).

Figure 2.4: Configurations for a SMI with two items.

proposes to add verification of exactly-one condition for SMIs on top of the variant constraints and IURs, and to use such verification every time the configuration data changes.

Beyond verification: authoring Item Usage Rules using partial configurations

When authoring IURs for a SMI, it is necessary to ensure that each valid configuration will have exactly one item from the SMI. This can be ensured by verifying the exactly-one property, as described above in Section 2.2.1. However, we can also consider a systematic way to create IURs that guarantees the exactly-one condition. A systematic way to use valid partial configurations as a basis for IURs is presented in Paper 1. Valid partial configurations for a given set of families do not coincide, since two different partial configurations for a set of families will never result in the same complete configuration. Thus, IURs based on valid partial configurations will make sure that items do not overlap. Valid partial configurations could also reveal some configurations that are valid, but must be forbidden, since there is no item to assign to them. This creates an iterative process. From the variant constraints the valid partial configurations are computed, from analysis of the valid partial configurations and IURs more variant constraints are potentially created, and the process repeats. This workflow is illustrated in Figure 2.5.

Computing valid partial configurations is a computationally difficult task. One way to enumerate all partial configurations and check each of them for validity, but even to check whether a single partial configuration is valid, it is necessary to take into account all possible extensions of that configuration and see if any of them satisfies all the constraints. Only checking a partial configuration against each constraint in isolation is not enough. For example in Table 2.1, if we take partial configurations involving

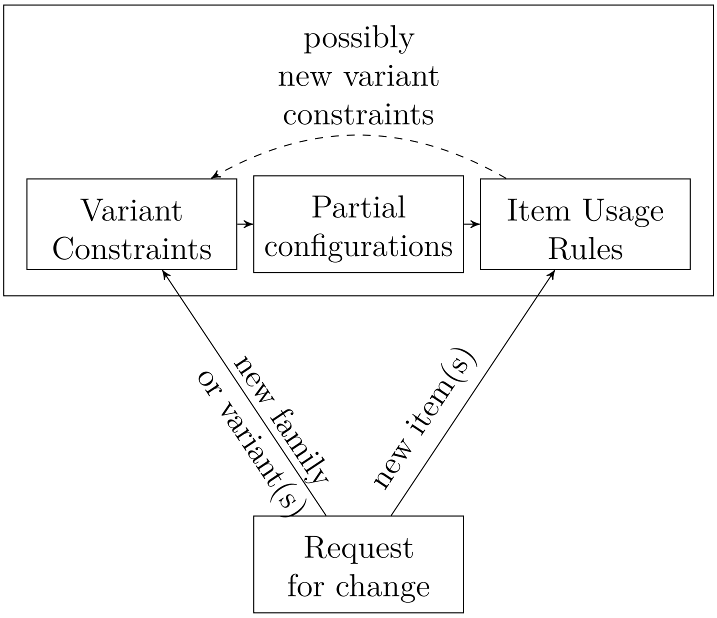


Figure 2.5: Workflow when using partial configurations to create item usage rules.

only *Volume* and *Turbo*, there is no direct constraint that connects these two families. Thus, it might seem that any combination of them is allowed. However, the partial configuration $\{$ *Volume=1.2*, *Turbo=Yes*$\}$ is not valid, because there are constraints connecting *Volume* with *Sport* and *Sport* with *Turbo*. This example illustrates that to verify properties of a subset of families it might be necessary to take into account other families as well. There could be a huge number of ways to extend a partial configuration to a complete one. For example, if there are 100 families, and a function group consists of two families, it is necessary to try all combinations of variants of the 98 remaining families, which will be 2^{98} if each family has only two variants; it is infeasible to explicitly check each of these configurations. Paper 3 introduces three efficient methods to compute valid partial configurations.

The number of valid partial configurations, and the IURs based on them, depend on the families used for partial configurations. It might be beneficial from a maintenance point of view to change the set of families used to create the IURs, but this might result in other problems, some of which are considered next.

2.2.2 Computational support for manual inspection

Apart from the automatic verification tasks introduced above, there are more tasks engineers might face, for example, authoring and modifying IURs, or discovering opportunities to improve the structure of the constraints. Such tasks can greatly benefit from computational support. For example, constraints can be rewritten automatically in different form depending on the needs of an engineer. Implicit relationships between items or families can also be made explicit automatically, as well as the effects of changes in constraints. These tasks are considered in this subsection.

Rewriting Item Usage Rules in terms of other families

When the IURs are already created, an engineer might want to rewrite an IUR in terms of other families, but keep intact the configurations for which the item is selected (Tidstam, Bligård, et al. 2012; Voronov, Åkesson, Tidstam, et al. 2012). This can be done in order to simplify and shorten an IUR, or to facilitate a better understanding of an IUR by showing it from "a different angle". Automatic rewriting from one set of families to another can contribute to the sustainability of the development by allowing different engineers to view IURs in their preferred ways.

Not all subsets of families are suitable for rewriting of an IUR. Adding more families is always safe. At worst, the size of the IUR will grow (possibly exponentially) due to the expansion of partial configurations into a number of more specific ones. Removing families, on the other hand, can lead to situations when it is ambiguous whether an item should be included or not, since when a family is removed, a number of more specific partial configurations can become a single less-specific partial configuration. Consider, for example IURs in Table 2.3a. If more families are added, a partial configuration for item *E12* splits into two more specific partial configurations, as illustrated in Table 2.3b. When some families are removed, two more-specific partial configurations become one less-specific partial configuration, as illustrated in Table 2.3c, making it impossible to create IURs that would uniquely define which item should be included.

Paper 2 introduces a way to verify that it is safe to delete a family from a given IUR, and, more generally, to verify that an alternative set of families is suitable for rewriting a given IUR.

Rewriting IURs is just one example of how constraints can be modified without changing their external behavior or meaning. The next subsection considers more such modifications.

Identifying refactoring opportunities

Software code *refactoring* is the process of changing a software system in such a way that the external behavior of the code is not altered, yet the internal structure of the code is improved (Fowler et al. 1999). The definition of refactoring can be extended to configuration constraints, where refactoring would be defined as changing the constraints without changing the valid configurations defined by the constraints, the reference configurations, or the forbidden reference configurations. The previously introduced rewriting of IURs in terms of other families can be regarded as one form of

Table 2.3: IURs.

(a) Initial IURs.

Families and variants				
Volume	Turbo	Sport	Fuel	*Item(s)*
1.6	yes	yes	gasoline	*E16T*
1.6	no	no	diesel	*E16D*
1.2	no	no	gasoline	*E12*

(b) IURs with extra family. Partial configuration for item *E12* split into two more-specific partial configurations.

Families and variants					
Volume	Turbo	Sport	City	Fuel	*Item(s)*
1.6	yes	yes	no	gasoline	*E16T*
1.6	no	no	no	diesel	*E16D*
1.2	no	no	yes	gasoline	*E12*
1.2	no	no	no	gasoline	*E12*

(c) IURs with families removed. Partial configurations for items *E16T* and *E16D* become one less-specific partial configuration.

Families and variants	
Volume	*Item(s)*
1.6	*E16T* or *E16D*
1.2	*E12*

refactoring. Paper 1 introduces more refactoring opportunities connected to the variant constraints. It should be noted that refactoring was also introduced for knowledge bases (Baumeister, Puppe, et al. 2004), which are related to configuration constraints.

Some configuration constraints might be implicit in the configuration data, or implied by other constraints. Some of such implicit constraints can be made explicit when refactoring, allowing to remove some of the presently-explicit constraints. It can also be discovered that an item has IURs, but there is no valid configuration that satisfies them, so the item is redundant and can be removed. Here is a number of refactoring questions:

- Does one item depend on another?
- Must two items always be selected together?
- Can two items ever be selected together?
- Is an item, or a variant, or a family redundant?

Answers to these questions can help engineers to understand the system and identify areas for refactoring.

What-if analysis: showing configurations that become invalid after introducing a new constraint

When adding a constraint, it might be difficult to foresee the effects of it. Unwanted side-effects may be introduced, for example, configurations that were allowed before adding the constraint may become forbidden. This may affect other teams concurrently working on the product. Thus, it is important to be able to assess the effects of adding, or removing, constraints to/from an existing constraint set. Paper 1 introduces a method to compute which valid configurations—either partial or complete—that are introduced or removed by a given modification of constraints.

2.3 Reconfiguration and Interactive Configuration

Reconfiguration is needed when a change in configuration constraints or user choices happens, and there is a need to find a new valid configuration (Crow and Rushby 1994; Männistö et al. 1999). Usually, such new valid configuration should be as close as possible to the original invalid configuration. The actual change that have to be made to the assignment is called a *repair* (Kreuz and Roller 1999; Felfernig, Friedrich, Schubert, et al. 2009; Schubert et al. 2011). Another relevant notion is *diagnosis* (Reiter 1987). Diagnoses can be used as a basis for reconfiguration (Crow and Rushby 1994), and this approach has been applied for product configuration, see, for example, (Felfernig, Friedrich, Jannach, et al. 2004). Corrective explanations (O'Callaghan et al. 2005) can also be considered as a reconfiguration. Reconfiguration is useful, for example, when dealing with aging and evolution of the product (Kreuz and Roller 1999; Manhart 2005; Falkner and Haselböck 2010; Friedrich et al. 2011), software upgrades

(Trezentos et al. 2010; Abate et al. 2012) and accommodating failures (Hadzic and Andersen 2005), as well as a feature in interactive product configuration.

Interactive product configuration (Gelle and Weigel 1996; Hadzic, Subbarayan, et al. 2004; Janota 2010) is a well-studied and widely implemented area of product configuration, and the most visible by customers. For example, a customer can configure a car on a manufacturer website, interactively choosing desired options while the configurator will ensure that the customer always selects only valid combinations of options. The user should have the possibility to choose values in any order, and the whole process should be *backtrack-free* and *complete*. *Backtrack-free* means that a user should always be able to finish the configuration procedure (select values for all variables) without a need to alter any of the earlier decisions. *Complete* means that if a configuration is valid, a user should have a way to achieve it. Interactive configuration sometimes also include a possibility to undo some of the earlier choices, we will call such actions *undo-actions*. Interactive configuration without undo-actions we will call *forward-only*. Another addition to reconfiguration can be *reconfiguration*, where an invalid configuration is changed into a valid one. During interactive configuration it can happen that the user wants to select a value that is not in the valid domain. In such a case, some of the previous choices have to be undone in order to make the new assignment valid. We will call such problem *interactive configuration with reconfiguration*. Reconfiguration can also be used outside of interactive configuration.

Enumeration of valid partial configurations, introduced in Paper 3, can be performed efficiently if there is an efficient interactive configurator. In the case of a forward-only configurator, each partial configuration can be checked for validity using the configurator. If a configurator supports undo-actions, the configurator can be used to generate a depth-first search tree from possible assignments, where each node of the tree will be an assignment of one value, and each terminal leaf of the tree will correspond to one partial configuration.

Chapter 5 introduces a novel method to efficiently represent configuration data needed for interactive configuration and reconfiguration.

2.4 Conclusions

This chapter introduced a number of problems in authoring and verification of configuration constraints that can be addressed by computational tools. Chapter 3 introduces formal methods and notation that can be used to describe these problems, and outlines how to apply the methods to the problems introduced here. Afterwards, Chapter 4 briefly introduces efficient algorithms and tools that can be used to solve large configuration instances.

Chapter 3

Introduction to Formal Methods

In computer science, formal methods are a particular kind of mathematically based techniques for systematic, rather than ad hoc, specification, design, and verification of software and hardware systems (Wing 1990). Formal methods for specification use mathematical notation to describe system requirements on some appropriate level of abstraction; formal specification can later guide the (non-formal) design process. Formal methods for verification check whether a model of a systems fulfills a given specification, usually by either examining the entire state-space of the system as done in Model Checking (Emerson and Clarke 1980; Clarke, Grumberg, et al. 2000), or by applying inference rules as done in Automated Theorem Proving (Robinson 1965; Duffy 1991). Formal design methods, called also *synthesis methods*, create a model of a system form a specification.

Formal methods have found the most widespread application in verification of hardware and software systems (Woodcock et al. 2009). In automotive product development formal methods were applied to embedded software, for example, to design and analyze a gear controller (Lindahl et al. 1998; Lindahl et al. 2001), to verify a car central locking system (Amnell and Jansson 2001), and to model and verify an infantry fighting vehicle rear ramp control system (Uckun 2011). However, if we include the use of all logic-based methods into the definitions of formal methods, then the use of many knowledge-based systems in product development—and especially the methods for their verification (Suwa et al. 1982; Gupta 1993; Preece et al. 1997; Tsai et al. 1999; Desharnais et al. 2011)—become relevant too. If we narrow the scope down to supporting automotive engineers in formally verifying correctness of the product configuration data, then the most relevant formal methods include efficient methods for representing configuration data and reasoning about it (Veron et al. 1999; Küchlin and Sinz 2000; Amilhastre et al. 2002; Pargamin 2002), as well as methods to formalize specific engineering problems and provide means to answer them (Amilhastre et al. 2002; Sinz et al. 2003; Astesana, Bossu, et al. 2010; Astesana, Cosserat, et al. 2010).

This chapter introduces constraint satisfaction, which is used in this thesis to model configuration data and to answer many verification questions. This chapter also introduces a synthesis method that is used in Chapter 5 to represent the configuration data for answering some of the configuration questions.

3.1 Constraint Satisfaction

We encounter constraints in everyday life, for example, when determining a seating arrangement for a dinner party, or when choosing a movie that a large group of friends will like. Constraint problems have three characteristic attributes: *variables*, their *domains* and *constraints*. *Variables* are objects that can take on a variety of values. The set of possible values for a given variable is called its *domain*. For the dinner party seating arrangement problem we may use chairs as variables. Each variable has the same domain – the set of all guests. *Constraints* impose limitations on the values that a variable, or a combination of variables, may be assigned. An example may be that the host and the hostess must sit at the two ends of the table, and that a feuding pair of guests should not sit next or opposite to each other. Many problems can be modeled in more than one way. For example, in the seating arrangement the guests may be the variables instead of the chairs, with the chairs being the domain of each variable. Such modeling difference is not important in the seating example due to a one-to-one correspondence between guests and chairs, but for other problems it might matter. A model that includes variables, their domains and constraints is called a *constraint problem*[1].

A *solution* or a *valid assignment* is an assignment of a single value from its domain to each variable such that no constraint is violated. A problem may have one, many, or no solutions. A problem that has one or more solutions is *satisfiable* or *consistent*. If there is no possible assignment of values to variables that satisfies all the constraints, then the problem is *unsatisfiable* or *inconsistent*.

Typical analysis of constraint problems is to determine whether a solution exists, finding one or all solutions, finding whether a partial instantiation can be extended to a full solution, and finding an optimal solution relative to a given cost function. Such tasks are referred to as *constraint satisfaction problems* (CSPs).

CSP for finite-domain variables belongs to the set of NP-complete problems (Cook 1971), and to date there is no algorithm known that can solve an arbitrary problem instance with a time complexity that is better than exponential in the size of the input (Hertli et al. 2011). However, if a problem instance possesses special structure (for example Horn formulas or 2-SAT, see Section 4.4), then the instance is polynomial-time solvable (Aspvall and Plass 1979; Dowling and Gallier 1984; Maaren 2000). The industrial problems that we tested do not belong to any of the known polynomial-time solvable classes. However, many solvers were still able to solve the industrial problems in a very short time (much faster than theoretically-predicted worst-case running time). Section 4.4 attempts to provide an explanation for that discrepancy.

The following section introduces the formal notation for constraint satisfaction.

[1] Freuder and Mackworth (2006) attribute the emergence of constraint satisfaction as a new paradigm within artificial intelligence and computer science to the 1965 paper by Golomb and Baumert "Backtrack programming" (Golomb and Baumert 1965).

Formal notation

Formally, a CSP is a triple $\mathcal{P} = \langle X, D, C \rangle$, where $X = \langle x_1, x_2, \ldots, x_n \rangle$ is an n-tuple of variables, $D = \langle D_1, D_2, \ldots, D_n \rangle$ is an n-tuple of corresponding finite domains, and $C = \{C_1, C_2, \ldots, C_J\}$ is a set of constraints. A constraint C_j is a pair $\left\langle R_{S_j}, S_j \right\rangle$, where R_{S_j} is a relation on the variables in $S_j = \mathit{scope}(C_j)$, and $\mathit{scope}(C_j) \subseteq X$ is a set of variables over which C_j is defined. In other words, R_{S_j} is a subset of the Cartesian product of the domains of the variables in S_j. In this thesis the relations are limited to propositional formulas over atomic propositions $x_k = v$ where $v \in D_k$.

A solution to the CSP $\mathcal{P}$ is an n-tuple $\mathcal{A} = \langle a_1, a_2, \ldots, a_n \rangle$ where $a_i \in D_i$ and each C_j is satisfied in that R_{S_j} holds on the projection of $\mathcal{A}$ onto the scope S_j. In a given task one may be required to find the set of all solutions, denoted by $\mathit{sol}(\mathcal{P})$, to determine if that set is non-empty, or just to find any solution, if one exists. If $\mathit{sol}(P)$ contains at least one solution, the problem $\mathcal{P}$ is said to be *satisfiable*, otherwise it is *unsatisfiable*.

A *complete assignment* to a CSP $\mathcal{P}$ is a function $f : X \to D$ which is defined for all $x_k \in X$. A complete assignment f is *valid* when its codomain (the target set of f, or the values for each variable) forms a solution. A *partial assignment* to $\mathcal{P}$ is a partial function $g : X \to D$ defined for variables $x_k \in Y \subseteq X$. We will write $\mathit{scope}(g) = Y \subseteq X$ to denote the set of variables of g. We will call a partial assignment *valid* if and only if it can be extended to a valid complete assignment, i.e. there exists a function h defined for $X \setminus Y$, such that codomains of g and h together form a solution.

One of the methods to encode constraints is by using *propositional formulas* (Chrysippus (c. 279 B.C. – c. 206 B.C.) is credited for the development of a coherent system of propositional logic (Johansen and Rosenmeier 1998)). A propositional formula is a formula that consists of a single propositional literal, or is a conjunction ("and", $\wedge$), disjunction ("or", $\vee$) or negation ("not", $\neg$) of propositional formulas. A propositional literal is either a positive or negative *atomic proposition*. An atomic proposition can be, for example, an expression saying that a variable takes a specific value, like "$x = 2$", or a Boolean variable. A *Boolean variable* (named after George Boole (1815-1864) who laid the foundations for an algebraic notation, which was later popularized by William Stanley Jevons (1835-1882) (Gardner 1958)) is a variable that have domain $\{0, 1\}$, or alternatively $\{$*(F)alse*, *(T)rue*$\}$. A propositional formula that consists only of Boolean variables is a *Boolean formula*. The problem of finding an assignment to the Boolean variables such that the Boolean formula evaluates to true is called *Boolean satisfiability problem* (SAT).

A propositional formula is said to be in *conjunctive normal form* (CNF) if it is a conjunction of *clauses*, where a *clause* is a disjunction of literals; for example, $(A \vee B \vee C) \wedge (B \vee D)$ is in CNF, while $(A \wedge B) \vee C$ is not in CNF. A Boolean formula in CNF can be represented using a set notation. A clause $l_1 \vee l_2 \vee \ldots \vee l_m$ is expressed as a set of literals $\{l_1, l_2, \ldots, l_m\}$. Moreover, a CNF $\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n$ is expressed as a set of clauses $\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Consider the following CNF over Boolean variables A, B, C, D:

$$(A \vee B \vee \neg C) \wedge (\neg A \vee D) \wedge (B \vee C \vee D).$$

Using set notation, it can be expressed as:

$$\{\{A, B, \neg C\}, \{\neg A, D\}, \{B, C, D\}\}.$$

A CNF Δ is *inconsistent* if it contains an empty clause: $\emptyset \in \Delta$. Moreover, if a CNF Δ contains no clauses, it is *consistent*: $\Delta = \emptyset$.

A *conditioning* of a CNF Δ on a literal L, denoted $\Delta|L$, is the process of replacing every occurrence of literal L by the constant *True*, replacing $\neg L$ by the constant *False*, and simplifying accordingly. All clauses that contain L become satisfied and can be removed from Δ. All clauses that contain $\neg L$ can be simplified by removing $\neg L$ from them (as it is *False* and no longer have any effect). All clauses that contain neither L nor $\neg L$ remain in $\Delta|L$ without change. *Unit resolution* is a conditioning procedure that takes the literal from a *unit clause*, where a *unit clause* is a clause that contains only one literal.

Having the notation covered, the next subsections introduce the basic building blocks of CSP and SAT solvers. Later, Chapter 4 and Paper 4 provide more methods to handle industrial problem instances.

3.1.1 Basic constraint satisfaction solver

A general algorithm for CSP solvers is illustrated by the procedure CSP-Solve in Algorithm 3.1 (Apt 2003). The algorithm is parameterized by the procedures Preprocess, Propagate-Constraints, Happy, Atomic, Split and Proceed-by-Cases. The Proceed-by-Cases procedure recursively invokes CSP-Solve.

Algorithm 3.1 CSP-Solve

```
Preprocess
Propagate-Constraints
if not Happy
  if Atomic
    done
  else
    Split
    Proceed-by-Cases
```

The first procedure in the algorithm is Preprocess, which prepares the problem for solving. It can convert a problem formulation to the form suitable for the solver or perform simplification of the problem to speed up the future solving process.

The procedure Happy checks whether the goal for the initial CSP has been achieved. The goal depends on the applications, but the most common goals are:

- some solution has been found,
- all solutions have been found,
- an inconsistency has been detected,

- an optimal solution with respect to some objective function has been found.

Atomic checks whether it is possible to split the current problem into smaller ones. If a solution or inconsistency has been found, the problem cannot be split.

The Split procedure divides the current problem into two or more subproblems, with the union of subproblems equivalent to the current problem. Splitting can be done on the domains of the variables or on the constraints. For example, if branching is done on a variable with domain size of three, then it is possible to split the problem into three sub-problems, each of which will have to deal with a single value for the variable. As an example of splitting on a constraint, consider a constraint that is a disjunction (OR). Such disjunction constraint can be split into multiple parts, with each subproblem having only one disjunct from the complete disjunction; if the subproblem for any of the disjuncts is satisfiable, then the original problem itself is satisfiable.

Proceed-by-Cases recursively invokes CSP-Solve for each subproblem (case) produced by the Split procedure. The cases are usually treated in a depth-first manner, turning the whole solving procedure into a backtracking depth-first search (Golomb and Baumert 1965). It starts at a root node and proceeds to the first descendant. The process is repeated until either the descendant node is a leaf or an inconsistency is found. If a leaf is a valid solution, the search terminates. Otherwise, the search backtracks to the previous node and continues by selecting the next unexplored descendant. If the search backtracks to the root node when all its descendants have already been explored, the search terminates declaring absence of solutions. The search can backtrack more than one level at a time, which is called non-chronological backtracking (Stallman and Sussman 1977; Bruynooghe 1981). It is also possible to remember the reason for the conflict in order to not repeat such partial assignments in the future (Stallman and Sussman 1977; R. Davis 1984; Dechter 1986; Dechter 1990).

The Propagate Constraints procedure reduces the search tree by inferring information from existing constraints. The simplest form is *node consistency*: if there is a unary constraint (i.e. defined only over one variable), then all values that do not satisfy the constraint should be removed from the domain of the variable. *Arc consistency* (Mackworth 1977) generalizes this to binary constraints: for each value of a variable in a binary constraint there should be a value in the domain of the other variable, such that the resulting pair satisfies the constraint. *Path consistency* (Montanari 1974) uses implicit induced constraints on triples of variables, i.e. considers all paths of size two among binary constraints to prune domains. *Generalized arc consistency* is an extension of arc consistency to constraints with more than two variables in their scope. A variable is generalized arc consistent with a constraint if every value of the variable can be extended to all the other variables of the constraint in such a way that the constraint is satisfied (Freuder 1978).

The CSP architecture presented in Algorithm 3.1 is very general and allows solving a variety of problems. SAT-solvers, on the other hand, can be seen as an extreme specialization of the CSP algorithm; they are considered in the next section.

3.1.2 Basic Boolean satisfiability solver

The basic procedure for Boolean satisfiability solving was presented by M. Davis and Putnam (1960), and later modified into a more memory-efficient search procedure by M. Davis, Logemann, et al. (1962). The resulting algorithm—commonly referred to as *DPLL*, by the initials of its authors—is given in Algorithm 3.2.

Algorithm 3.2 DPLL(Δ)

```
1  input: CNF $\Delta$
2  output: SATISFIABLE or UNSATISFIABLE
3  $\Delta' = $ Unit-Resolution($\Delta$)
4  if $\Delta' = \emptyset$
5    return SATISFIABLE
6  else if $\emptyset \in \Delta'$
7    return UNSATISFIABLE
8  else
9    choose a literal $L$ in $\Delta'$
10   if DPLL($\Delta'|L$) = SATISFIABLE
11     return SATISFIABLE
12   else if DPLL($\Delta'|\neg L$) = SATISFIABLE
13     return SATISFIABLE
14   else
15     return UNSATISFIABLE
```

It is easy to see similarities between Algorithms 3.1 and 3.2. The unit resolution in line 3 of Algorithms 3.2 is a simple form of constraint propagation, which propagates all clauses that consist of a single literal (unit clauses) by conditioning the problem on those literals. Lines 4-7 of Algorithm 3.2 are a merge of both tests for Happy and Atomic. Split is implemented by choosing a literal in line 9 of Algorithms 3.2, and Proceed-by-Cases is implemented by a recursive call to the DPLL procedure on a conditioned CNF.

The original DPLL algorithm presented here was invented half a century ago. Modern efficient implementations of SAT algorithm depart from the original DPLL in various ways. An overview of some of the most important modifications is given in Chapter 4 and Paper 4. Modern SAT algorithms have very good performance, and they are often used to solve problems that were modeled as CSP (Prestwich 2009), by converting CSP to SAT. Such conversion is briefly introduced in the following section.

3.1.3 Encoding constraint satisfaction problems as Boolean satisfiability problem

Often problems are modeled using CSP, and then converted to SAT to take advantage of many efficient SAT tools (Prestwich 2009). Wide range of efficient tools use a common encoding of Boolean satisfiability problem called DIMACS CNF format. The format requires the problem to be described in terms of only Boolean variables, and the constraints have to be CNF clauses; a CSP with finite-domain variables and propositional constraints can be converted to (Boolean) SAT with CNF constraints with a polynomial increase of the problem size.

> The DIMACS CNF format is as following. The file starts with zero or more comment lines, indicated by the character `c` at the beginning of the line. After the comment lines, there is a “header string” `p cnf n m` that indicates that the instance is in CNF format; n is the number of variables; m is the number of clauses. The header string is followed by the clauses. Each clause (a disjunction of literals) is encoded as a sequence of literals, where each literal is represented by a number between $-n$ and n, the clause ends with 0 on the same line; a clause cannot contain the opposite literals i and $-i$ simultaneously. A positive number i denotes the corresponding variable x_i. A negative number $-i$ denotes the negations of the corresponding variable $\neg x_i$. An example file content for the formula $(x_1 \vee \neg x_5 \vee x_4) \wedge (\neg x_1 \vee x_5 \vee x_3 \vee x_4) \wedge (\neg x_3 \vee \neg x_4)$ can look as the following:
>
> ```
> c
> c start with comments
> c
> p cnf 5 3
> 1 -5 4 0
> -1 5 3 4 0
> -3 -4 0
> ```

To provide a brief introduction to the conversion from CSP to SAT, the encoding of finite-domain variables using Boolean variables, and encoding of arbitrary propositional formula using CNF is given in the following two sections.

Encoding finite-domain variables using Boolean variables

Two of the most widespread encodings of finite-domain variables using Boolean variables are *direct encoding* (Kleer 1989; Walsh 2000) (also called *one-hot encoding* (Biere and Kunz 2002)) and *log encoding* (Iwama and Miyazaki 1994; Walsh 2000). Direct encoding assigns one Boolean variable to each value. The Boolean variable indicates if the value is assigned to the variable or not. Thus, the number of Boolean variables is equal to the number of values in the domain. Log encoding assigns a unique integer in the interval from 0 to $n-1$ to each value in the domain, where n is the size of the domain. In this case $\lceil \log_2 n \rceil$ Boolean variables are needed to encode the values.

Direct encoding requires extra constraints to encode that exactly one value can be selected for each variable. Exactly-one constraints can be split into two constraints: at-least-one and at-most-one. At-least-one constraints can be encoded using a single disjunction of n literals. At-most-one constraints, on the other hand, is more difficult to encode. The simplest encoding of an at-most-one constraint would require n^2 disjunctions of the form $(\neg x_i \vee \neg x_j), i \neq j$, where x_i and x_j belong to the domain of the variable; this encoding is called *quadratic* in this thesis due to the number of extra constraints needed. Other encodings exist (Sinz 2005; Frisch and Giannaros 2010; Ben-Haim et al. 2012; Manthey et al. 2012), including generalizations of at-most-one to at-most-k called *cardinality constraint*. One of the encodings is *ladder encoding*

(Gent and Nightingale 2004), which introduces a linear number of new constraints and a linear number of extra variables. Ladder and quadratic encodings are used in this thesis to encode at-most-one constraints for direct encodings of finite-domain variables.

Log encoding does not require at-least-one and at-most-one constraints, as mutual exclusiveness is implicit in the encoding. However, to encode variables with domains of sizes not a power of 2, log encoding requires constraints to forbid spurious assignments that would correspond to integers between n and $2^{\lceil \log_2 n \rceil} - 1$. Despite the benefit of reduced number of variables needed by log encoding, the drawback of log encoding is that unit propagation on the log encoding is less effective than unit propagation on the direct encoding (Walsh 2000), which might result in worse SAT-solver performance.

Encoding propositional formulas into CNF

To convert a propositional formula to CNF, one can use the rules of Boolean algebra. The conversion can be exemplified by transforming formula $(A \wedge B) \vee \neg(C \wedge \neg D)$, where A, B, C, D are Boolean variables:

$$\begin{gathered}(A \wedge B) \vee \neg(C \wedge \neg D) = \\ (A \wedge B) \vee (\neg C) \vee D = \\ (A \vee \neg C \vee D) \wedge (B \vee \neg C \vee D)\end{gathered}$$

Logic transformations might result in exponential growth of the formula, for example, when converting Disjunctive Normal Form (disjunction of conjunctions) to CNF.

Another approach to convert a formula to CNF is to use Tseitin transformation (Tseitin 1968), which introduces auxiliary variables and results only in polynomial growth of the formula. For the example above, Tseitin transformation would introduce two new variables, X and Y, and the following extra formulas for the new variables:

$$\begin{gathered}X \leftrightarrow (A \wedge B) \\ Y \leftrightarrow (C \wedge \neg D)\end{gathered}$$

where $P \leftrightarrow Q$ stands for $(P \rightarrow Q) \wedge (Q \rightarrow P)$. Also, $P \rightarrow Q$ stands for $(\neg P) \vee Q$. Then the original formula can be replaced by the following clauses:

$$\begin{array}{ll}
X \vee \neg Y & \%\ \textit{Original formula} \\
(\neg X) \vee A & \%\ X \rightarrow A \\
(\neg X) \vee B & \%\ X \rightarrow B \\
(\neg A) \vee (\neg B) \vee X & \%\ (A \wedge B) \rightarrow X \\
(\neg Y) \vee C & \%\ Y \rightarrow C \\
(\neg Y) \vee (\neg D) & \%\ Y \rightarrow \neg D \\
(\neg C) \vee D \vee Y & \%\ (C \wedge \neg D) \rightarrow Y
\end{array}$$

For smaller formulas, like the one above, Boolean algebra transformations give smaller CNF, but for bigger formulas, Tseitin transformation is necessary to avoid exponential growth of the formula. Consider, for example, converting $(A \land B \land C) \lor (D \land E \land F) \lor (G \land H \land I)$, which would result in 27 clauses using Boolean algebra, and only in 13 clauses and 3 extra variables using Tseitin transformation.

Satisfiability problems are well-suited for answering verification questions. For example, SAT often serves as a target representation for answering verification questions in model checking (Biere, Cimatti, et al. 1999). However, some problems might benefit from other formal methods such as synthesis. One synthesis method, supervisory control synthesis, is introduced in the next section. Furthermore, there is a relation between verification and synthesis, and in Paper 4 we introduce the first encoding of some of supervisory synthesis problems as SAT problems. The next section briefly introduces supervisory control theory and synthesis problems.

3.2 Synthesis using Supervisory Control Theory

Verification answers only *yes* or *no* to the question whether a specification is fulfilled or not by the system, possibly with a proof for *yes*, and a counter-example for *no*. But it is possible to go further. For example, the supervisory control theory (SCT) (Ramadge and Wonham 1989; Cassandras and Lafortune 2008) provides a way to synthesize (automatically compute) a safety device, called a *supervisor*, that restricts the behavior of an uncontrolled process, the *plant*, in such a way that the desired behavior of the controlled system, a *specification*, is fulfilled. The synthesized supervisor controls the plant so that it always stays within the limits defined by the specification, by dynamically disallowing the generation of events that might lead to a behavior outside the specification, see Figure 3.1.

SCT includes a certain type of "controllability". The supervisor is mainly a safety device that prevents the plant from executing events that would take the controlled system outside the specified behavior. However, not all events can be prevented from occurring; some events are *uncontrollable*, and the supervisor must never (try to) disable any of the uncontrollable events, since these events may be spontaneously generated by the plant.

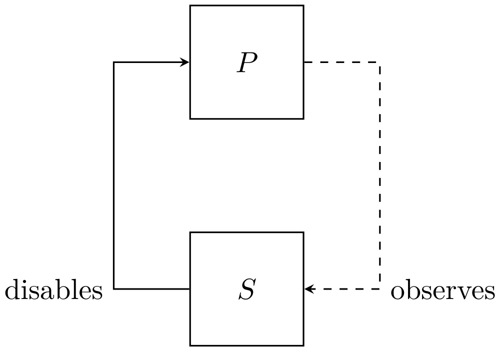


Figure 3.1: Plant P supervised by a supervisor S. P spontaneously generates events from a set allowed by S. The generated events are observed by S, which generates a new set of disabled events for P.

In addition to controllability, it is desired for the supervisor to be *non-blocking*. A non-blocking supervisor guarantees that at least one *marked* state is reachable from any state that the closed-loop system (plant and supervisor) is allowed to reach. Marked states typically represent (sub-)tasks that the system must always be able to finish.

Ramadge and Wonham (1987) have shown that for a given plant and a controllable and non-blocking with respect to the plant specification, there always exists an optimal supervisor guaranteeing that the specification will not be violated, while at the same time allowing the system to always fulfill at least one of its defined (sub-)tasks. The optimality criterion for the supervisor is to restrict the given plant as little as possible. Such a supervisor is said to be *maximally permissive* or equivalently *minimally restrictive*.

Synthesis can be viewed as a series of verification tasks, where the process model (the plant) allows an automatic alteration of the suggested, and negatively verified, supervisor candidate. The original specification can be viewed as the first supervisor candidate; if it is verified to be correct (controllable and non-blocking) then no further processing is necessary. Otherwise, a new supervisor candidate will be created by removing undesired behavior from the previous supervisor candidate, and the process repeats. Thus, by construction, a synthesized supervisor will always be verified to be correct.

SCT and CSP have some similarities. In CSP, given a set of constraints, the task is to find a set of satisfying assignments. In SCT, given a model of a plant, plus a given set of constraints (specification), the task is to find another model (supervisor), such that this model interacting with the plant satisfies the constraints.

From complexity point of view, answering whether there exist a supervisor with respect to a modular finite-state automata system that guarantees reaching a marked state is NP-hard (Gohari and Wonham 2000), and verifying the admissibility of a single supervisor with respect to a modular finite-state automata system is PSPACE-complete (Rohloff and Lafortune 2005), while CSP for finite-domain variables and SAT are NP-complete (Cook 1971).

Modeling formalism

As a modeling formalism for supervisory control theory problems, *finite state automata* may be used. Formally, a deterministic finite state automaton (*FSA*), denoted by A, is defined as a five-tuple (Cassandras and Lafortune 2008):

$$A = \langle Q, \Sigma, f, q_0, Q_m \rangle \,,$$

where Q is the finite set of *states*; Σ is the finite set of *events*, i.e. the *alphabet*, associated with the transitions in A; $f : Q \times \Sigma \to Q$ is the partial *transition function*: $f(q, \sigma) = p$ means that there is a transition labeled by event σ from state q to state p; $q_0 \in Q$ is the *initial* state; and $Q_m \subseteq Q$ is the set of *marked* states.

The transition function can be written in infix notation; for example, $q \xrightarrow{\sigma} p$ denotes a transition from state q to state p associated with event σ. This notation is further extended in the natural way to sequences of events of finite length.

The transition function, f, is partial and thus not all events are defined from all states. The *active event function* $\Gamma(q)$ denotes the set of all events σ for which $f(q,\sigma)$ is defined; this set is called the *active event set.* If $\sigma \in \Gamma(q)$ we say that the event σ is *enabled* in state q. The active event function is implicitly defined from the transition function f.

A state q_i is called *reachable* if there exists a sequence of events that leads from the initial state (q_0) to state q_i.

Typically, a model of a plant or a specification consists of different submodels focusing on different aspects. A specific composition operator *parallel composition* (also called *synchronous composition*), see for example (Cassandras and Lafortune 2008), may be used to compose a full model (plant or specification) from multiple submodels.

Parallel composition models interaction. In a parallel composition of automata, an event σ can occur only from a joint state where each automaton has σ enabled. Let $A_1 = \langle Q^1, \Sigma^1, f^1, q_0^1, Q_m^1 \rangle$ and $A_2 = \langle Q^2, \Sigma^2, f^2, q_0^2, Q_m^2 \rangle$. The parallel composition of A_1 and A_2 is the automaton

$$A_1 || A_2 = \left\langle Q^1 \times Q^2, \Sigma^1 \cup \Sigma^2, f^{1||2}, (q_0^1, q_0^2), Q_m^1 \times Q_m^2 \right\rangle .$$

The transition function, $f^{1||2}$, is defined as

$$f^{1||2}((q^1,q^2),\sigma) := \begin{cases} (f^1(q^1,\sigma), f^2(q^2,\sigma)) & \text{if } \sigma \in \Gamma^1(q^1) \cap \Gamma^2(q^2) \\ (f^1(q^1,\sigma), q^2) & \text{if } \sigma \in \Gamma^1(q^1) \backslash \Sigma^2 \\ (q^1, f^2(q^2,\sigma)) & \text{if } \sigma \in \Gamma^2(q^2) \backslash \Sigma^1 \\ \textit{undefined} & \textit{otherwise.} \end{cases}$$

$\Gamma^{1||2}$ follows from the definition of $f^{1||2}$ and is given by

$$\Gamma^{1||2}(q^1,q^2) = \left(\Gamma^1(q^1) \cap \Gamma^2(q^2)\right) \cup \left(\Gamma^1(q^1) \backslash \Sigma^2\right) \cup \left(\Gamma^2(q^2) \backslash \Sigma^1\right) .$$

Only the reachable states are of importance during analysis; thus it is common to keep only the reachable subset of $Q^1 \times Q^2$ in the composition. The parallel composition operator is associative and commutative (Cassandras and Lafortune 2008), and can thus be extended in a straightforward way to compose an arbitrary number of automata.

Controllability Let P and S be two automata. Let Σ_u be the set of uncontrollable events and Σ^S be the alphabet of S. A state $(q^P, q^S) \in Q^P \times Q^S$ in the synchronized automaton $P||S$ is controllable if the following statement holds:

$$\Sigma^S \cap \Sigma_u \cap \Gamma(q^P) \subseteq \Gamma(q^S), \textit{ assuming } \Sigma^S \subseteq \Sigma^P .$$

Uncontrollable states are the states in $P||S$ where P enables an uncontrollable event, but S disables the same event (i.e. S has the event in its alphabet, but does not have the event in the active event set of the current state). Let P be the plant and S be a specification, and Σ_u be the set of uncontrollable events. S is controllable with respect to P and Σ_u if all reachable states of $P||S$ are controllable. It is known (Ramadge and Wonham 1987) that for a given specification and plant, a supervisor, which guarantees that the entire specification can be achieved, exists if and only if the specification is *controllable* with respect to the plant. This means that the specification must be such that it can be enforced without having to (try to) disable any uncontrollable events. If the original specification is not controllable with respect to the plant, a controllable sub-behavior of the specification has to be computed. It is known that the union of all controllable sub-behaviors of a specification with respect to a plant is also controllable thus a *supremal controllable sublanguage* exists. The supervisor's task is to restrict the behavior of the plant such that the supremal controllable sublanguage is achieved. If no controllable sublanguage exists, which implies that the supremal controllable sublanguage is empty, then no supervisor exists; in such a case it is necessary to change the plant or the specification.

Non-blocking Let $A = \langle Q, \Sigma, f, q_0, Q_m \rangle$ be an automaton. A reachable state $q \in Q$ is said to be non-blocking if there is a path from q to some marked state:

$$\exists s \in \Sigma^*, \text{ such that } \exists q_m \in Q_m \text{ where } q \xrightarrow{s} q_m$$

An automaton is said to be *non-blocking* if all of its reachable states are non-blocking. In words, the system is non-blocking if from any reachable state there is a path to some marked state.

Deadlocks A reachable state $q \in Q$ is a deadlock state if there is no transition leaving the state:

$$\Gamma(q) = \emptyset.$$

In many applications it is desirable to have no deadlocks, but in some applications deadlocks arise naturally. For example, if a manufacturing system has to produce a fixed number of parts, then, after those parts are produced, the system is in a desired deadlock. Such deadlocks at the end of a set of tasks can often be avoided either by adding a transition back to the initial state, or by adding self-loops at the desired final states, or by making desired final states marked. The remaining (non-marked) deadlocks would usually indicate some undesired behavior of the system, and the need for controlling the system. Thus, the synthesis procedure should make sure that we the closed-loop system is not allowed to reach non-marked deadlock states.

Supervisor synthesis algorithm A simple algorithm for synthesis of a non-blocking and controllable supervisor (Cassandras and Lafortune 2008) starts by computing the first supervisor candidate as the synchronous composition of the plant and specification automata. Then uncontrollable and blocking states are removed from the candidate repeatedly until there are no more uncontrollable or blocking states left. This way, the resulting supervisor is controllable and non-blocking by construction.

Example: applying Supervisory Control Theory to a robot and a machine

As an example, let us consider one robot and one machine as shown in Figure 3.2. The plant consists of the two automata shown in Figure 3.3a and 3.3b that model the robot and the machine. It is assumed that the robot will spontaneously release a manufacturing part (corresponding to the *put*-event) in the machine after it has picked it up. Thus, the put-event is uncontrollable. Uncontrollable events are prefixed with an exclamation mark (!) in the figure. We would like the system to fulfill the following specification (Figure 3.3c): after each *(!)put* event there should follow event *load* followed by *unload_A*. This guarantees that the robot will not put a new part into the machine before the machine has consumed the current part. It also restricts the machine to use output buffer A only. In this example, the plant, P, is given by $Robot||Machine$ and the specification S consists of a single automaton. In general, both a plant and a specification can consist of multiple sub-plants and sub-specifications.

The full synchronous composition of R, M and S is shown in Figure 3.4. It contains two uncontrollable states, which the synthesis algorithms removes to produce the four-states supervisor. The supervisor resulting from synthesis algorithm for Robot, Machine and Specification example is shown in Figure 3.5.

The example illustrated the basic concepts of SCT, including modeling the system using finite state automata and synthesizing a controllable and non-blocking supervisor. Chapter 5 shows how to use SCT to solve product configuration problems.

3.3 Conclusions

To summarize, this chapter introduced basic mathematical concepts used in this thesis: CSP, SAT and SCT. CSP and SAT are used to represent configuration problems in Papers 1, 2 and 3. Approaches for solving large satisfiability problems arising from

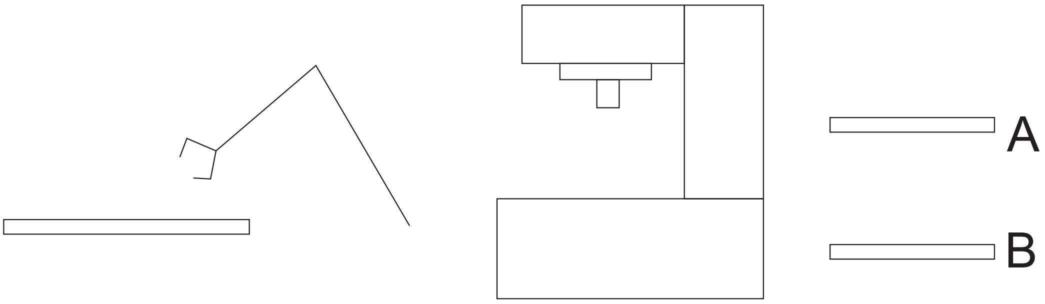
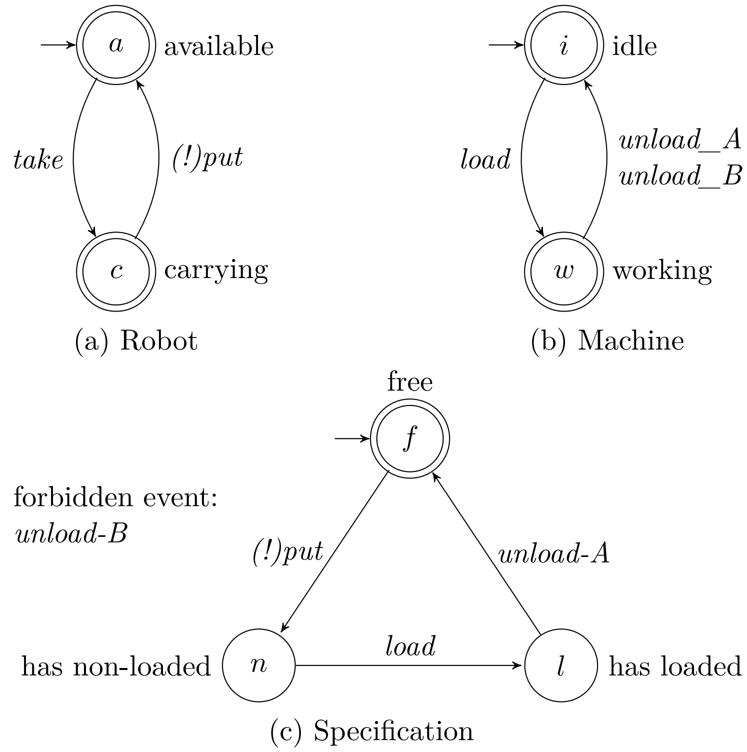


Figure 3.2: A robot and a machine. The robot takes parts from the input buffer and puts them on the machine. The machine loads the part brought by the robot, and after processing unloads it to the output buffer A or B.



(a) Robot

(b) Machine

(c) Specification

Figure 3.3: Automata models of the plant, consisting of the robot and the machine, and the specification. The exclamation mark (!) before an event name indicates that the event is uncontrollable. The alphabets are as following: $\Sigma^{Robot} = \{take, !put\}$, $\Sigma^{Machine} = \{load, unload\_A, unload\_B\}$, and $\Sigma^{Spec} = \{!put, load, unload\_A, unload\_B\}$. Note that the specification has no transition labeled *unload_B*, but this event is in the alphabet of the specification, thus *unload_B* is never allowed by the specification.

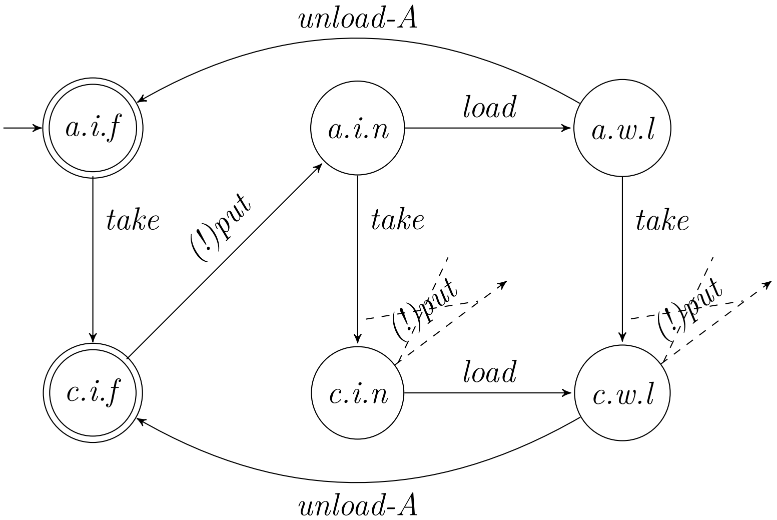


Figure 3.4: The synchronous composition $R||M||S$, with uncontrollable states $s.i.n$ and $c.w.l$ denoted by dashed crossed-out arrows pointing away from the states.

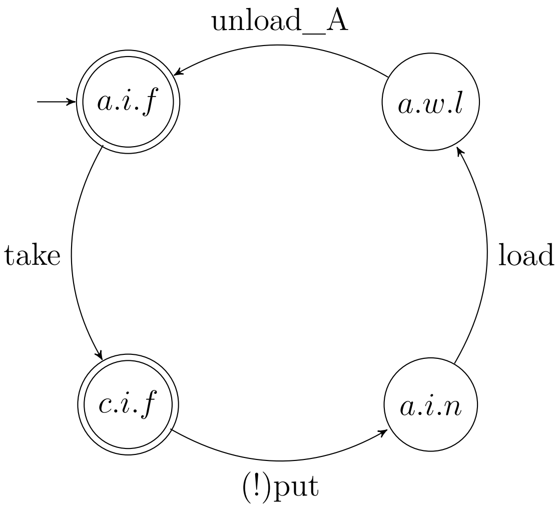


Figure 3.5: Minimally restrictive supervisor for the robot and the machine from Figure 3.3

large configuration problems are presented in Chapter 4. SCT is used to represent configuration problems in Chapter 5. Using SAT to encode SCT problems is presented in Paper 4. The next chapter looks at efficient methods to solve large problem instances arising in product configuration.

Chapter 4

Solving large-scale problems

Chapter 2 introduced some challenges in product development, and Chapter 3 introduced some techniques for formalizing product configuration problems and outlined how these techniques can be used to solve the problems in Chapter 3. This chapter briefly introduces different computational methods that were suggested for working with large-scale configuration problems, and empirically evaluates some of them. The chapter starts with the modern SAT-solvers and the key features that improved their performance compared to the basic DPLL algorithm presented in Section 3.1.2. Then, knowledge compilation methods are introduced, which separate the computations into an "expensive" offline phase and a "cheap" online phase, providing the ability to reuse the offline computation for many online queries about the same data. The chapter goes on to compare several implementations of known methods on industrial configuration data. The benchmark reveals that SAT-solvers provide answers in a much shorter time than the theoretically predicted worst-case running time, and in the last section of this chapter we investigate this discrepancy (but we have not found an answer yet, and further investigations are needed).

4.1 Search-based Boolean Satisfiability Solvers

A search-based Boolean Satisfiability Solver, or a SAT-solver, has to, given a formula in CNF, find an assignment to the Boolean variables such that the formula evaluates to true, if possible. An equivalent formulation is to say that *each clause* should have at least one literal that is true under a certain assignment. Such a clause is said to be *satisfied*. If there is no assignment satisfying all clauses, the CNF formula is said to be *unsatisfiable*. Propositional formulas that are not in CNF can be transformed into CNF using the methods discussed in Section 3.1.3.

A simple and complete SAT algorithm can be achieved by a standard backtracking search. During the search, a *partial assignment* is maintained, where some variables are assigned to 0 or 1 (alternatively to *True* or *False*), and others are unassigned. The backtracking search picks a variable, assigns it to 0 or 1 (this is called a *decision*), and repeats the procedure for the subproblem. If no solution is found, the other value is tried. If a conflict is found (all literals in a clause are false), there is no need to branch further. If all clauses have at least one literal true, then a solution has been found.

Sometime partial assignments can force the values of other variables, which is called *constraint propagation.* Consider the following three CNF clauses:

$$(\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee d)$$

If a partial assignment is $a = 1$, then there is no other way to satisfy the first clause other than assign $b = 1$. In the same way the assignment is propagated to $c = 1$. Theses two assignments, in turn, will lead to $d = 1$. Such propagation takes place when all but one literal of a clause are false, and can be implemented very efficiently (Moskewicz et al. 2001).

Constraint propagation can be run after every assignment, resulting in the DPLL algorithm presented in Section 3.1.2, which until the inception of modern SAT solvers was the predominant approach to SAT.

The major differences of modern solvers from DPLL can be summarized as following:

- It is not a recursive procedure. Instead, an explicit stack of assignments is used for backtracking.
- When a conflict is found, the solver learns from it. From each conflict, a *conflicting clause* is derived and added to the set of clauses. The conflict clause strengthens the propagation without changing the satisfiability of the formula.
- Backtracking is no longer restricted to return to the previous decision. If the last k decisions were irrelevant to the conflict, all of them are undone, together with their propagated assignments.

These differences resulted in a new name for modern SAT-solvers: Conflict-Driven Clause Learning (CDCL) solvers (Marques-Silva and Sakallah 1996; Marques-Silva, Lynce, and Malik 2009).

The algorithm of a modern CDCL SAT-solver can be summarized as in Algorithm 4.1 (Claessen et al. 2008).

Algorithm 4.1 CDCL-SOLVE

```
forever:
    PROPAGATE-CONSTRAINTS
    if NO-CONFLICT
        if NO-UNASSIGNED-VARIABLE then return SAT
        MAKE-DECISION
    else
        if NO-DECISIONS-WERE-MADE then return UNSAT
        ANALYZE-CONFLICT
        UNDO-ASSIGNMENTS
        ADD-LEARNED-CLAUSE
```

Another important part of a solver is how it makes decisions. The most successful heuristic so far tries to bias decisions towards variables that were recently involved in the conflicts. This helps to find as many conflicts as possible in a small region of the search space, which should result in a set of short clauses that capture the reasons for the conflicts better than the original clauses.

The mentioned features of the SAT-solvers' algorithms and well-selected and tuned heuristics have made SAT-solvers very powerful computing tools successfully applied to many practical problems (Marques-Silva 2008). In particular, SAT-solvers were successfully used for product configuration (Küchlin and Sinz 2000; Sinz et al. 2003; Janota 2008; Janota 2010).

Often, especially in many product configuration problems, the same data is used to answer many queries, which makes starting the search "from scratch" less desirable. Instead, it is possible to keep learned clauses as in incremental solving (Een and Sörensson 2003), or to compile the whole problem into a tractable representation, as done in the knowledge compilation approaches considered in the next section.

4.2 Knowledge compilation methods

Knowledge compilation is a family of approaches that addresses intractability of many Artificial Intelligence problems. A propositional model is compiled in an offline phase in order to support some online queries in polytime. Many knowledge compilation methods exist, see, for example, (Cadoli and Donini 1997; Darwiche and Marquis 2002) for reviews, but the most widespread knowledge compilation method is Binary Decision Diagram, which is considered in the following subsection.

4.2.1 Binary Decision Diagrams

Binary Decision Diagrams (Lee 1959; Akers 1978; Bryant 1986) can be used to efficiently find a satisfying assignment to SAT. A Boolean formula can be represented as a decision tree, where nodes of the tree represent variables, and edges of the tree represent values of the variables. An example tree for the function $a \wedge b$ is shown in Figure 4.1a. Each decision node is labeled by a Boolean variable and has two child nodes called the *low child* and the *high child*, respectively. The edge from a node to a low (high) child represents an assignment of the variable to 0 (1). Terminal leafs represent the final value that the function will take when variables are assigned the values found on the path from the root of the tree to the leaf. Since the function is Boolean, there are only two unique terminal leafs: 0 and 1. By keeping only two unique leafs (and calling them 0-terminal and 1-terminal), the tree will become a rooted, directed, acyclic graph, which consists of decision nodes and two terminal nodes 0-terminal and 1-terminal. This graph is called a Binary Decision Diagram (BDD). A BDD is called *ordered* if different variables appear in the same order on all paths from the root. A BDD is said to be *reduced* if the following two rules have been applied to its graph: (i) any isomorphic subgraphs are merged, and (ii) any node whose two children are isomorphic is eliminated. In what follows, we will call such a reduced ordered BDD simply BDD. An example of such a BDD is shown in Figure 4.1b. In practice, BDDs are generated and manipulated in their fully reduced form, without ever building the decision tree. *Multi-valued decision diagrams* (MDDs)

(Kam et al. 1998) are a generalization of BDDs to variables with arbitrary finite-sized domains.

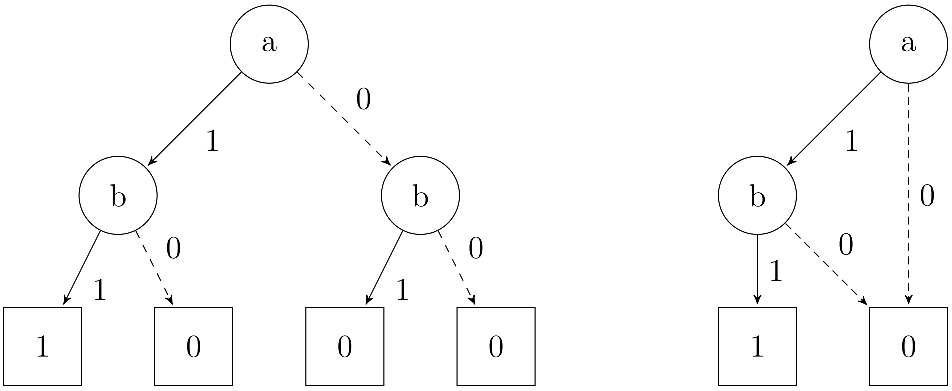
BDDs have several important benefits. First of all, it is a compact representation of a Boolean function, which saves space compared to some other representations. Secondly, it is easy to manipulate BDDs: all usual logical operations (AND, OR, NOT, NAND, etc.) can be performed directly on BDDs. A BDD with a fixed variable ordering is also a canonical representation of a Boolean function, which makes it easy to check functions for equivalences.

The disadvantage of BDDs is that they are sensitive to the variable ordering (Bryant 1986). The problem of choosing the best variable ordering is NP-complete (Bollig and Wegener 1996). An example of the difference between a good and a bad variable ordering for a BDD is shown in Figure 4.2.

Despite the fact that BDDs have been successfully used for product configuration (Hadzic, Subbarayan, et al. 2004; Subbarayan, Jensen, et al. 2004), for many practical problem instances BDDs can require a large amount of memory. Other knowledge compilation methods may provide more succinct representations of a problem instance at the expense of reduced number of supported polytime queries and transformations. Such knowledge compilation methods relevant to product configuration are outlined in the next subsection.

4.2.2 Other knowledge compilation methods

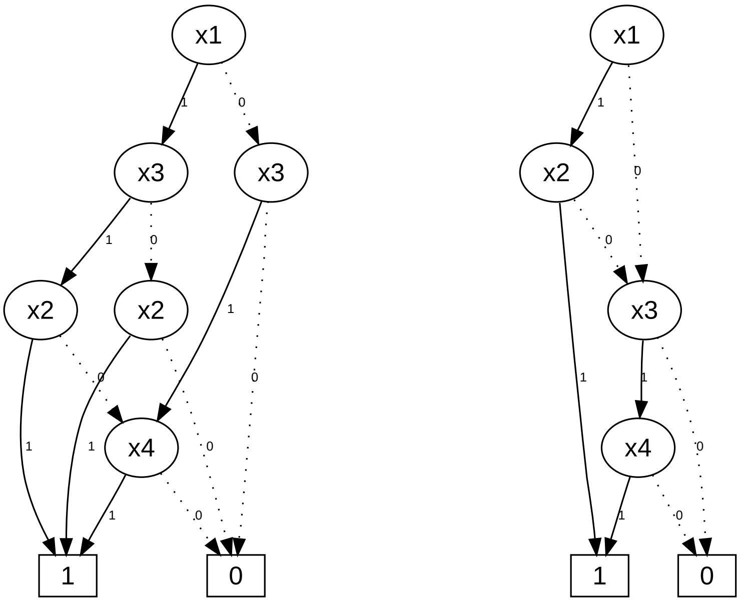
Darwiche and Marquis (2002) presented a summary of knowledge compilation methods (a "knowledge compilation map"), along with the properties of each method and polytime supported queries. Later, the map was extended with more methods, for example, Tree-of-BDDs (Subbarayan, Bordeaux, et al. 2007; Fargier and Marquis 2009). Among knowledge compilation methods, the most relevant to product configuration are Decomposable Negation Normal Form (Darwiche 1998; Darwiche 2001a), Cluster trees (Dechter and Pearl 1989; Pargamin 2002), Automata (Amilhastre et al. 2002), Multivalued Decision Diagrams (Kam et al. 1998; Hadzic and Hansen 2008), Tree-driven automata (Fargier and Vilarem 2004), AND/OR Multivalued Decision Diagrams



(a) Decision tree

(b) BDD

Figure 4.1: Decision tree and reduced ordered BDD for function $a \wedge b$.



(a) Bad variable ordering (b) Good variable ordering

Figure 4.2: Good and bad variable orderings for formula $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$.

(Mateescu and Dechter 2006) and Tree-of-BDDs (Subbarayan 2005). These data structures are briefly introduced below.

Decomposable Negation Normal Form

Decomposable Negation Normal Form (DNNF) (Darwiche 1998; Darwiche 2001a) is a data structure of which BDD is a special case. A propositional formula is in negation normal form (NNF) if and only if it is one of the following:

- a *literal*, which is a positive or negative *atomic proposition* (*atom*);
- a conjunction of formulas in NNF;
- a disjunction of formulas in NNF.

A formula in NNF is *decomposable* (DNNF) if and only if for any conjunction no atoms are shared by any conjuncts (in other words, for every conjunction the following must hold: for any two children of the conjunction, the sets of atoms do not overlap). A formula in NNF is *deterministic* (d-NNF) if for every disjunction, every pair of disjuncts is logically inconsistent (only one child of a disjunction can be true for any given assignment). A formula in NNF is *smooth* (s-NNF) if for every disjunction the set of atoms is equal to the set of atoms of each of its children (all children of a disjunction have the same sets of atoms). If a formula in DNNF is both smooth and deterministic, we write sd-DNNF.

DNNF is more succinct than BDDs and its compilation time is often shorter than that of BDDs (Subbarayan, Bordeaux, et al. 2007; Voronov, Åkesson, and Ekstedt 2011). DNNF supports smaller number of tractable operations than BDD (Darwiche and

Marquis 2002), while still allowing polynomial-time (in the size of the DNNF and/or the number of solution) counting of solutions (on sd-DNNF), existential quantification of atoms (forgetting) (Darwiche 1999) and solutions enumeration (Darwiche 2001b).

DNNF was proposed for use in product configuration for solutions (or valid configuration) counting to measure documentation maturity and estimate complexity of detecting errors in product configuration; these methods may be useful when comparing configuration data, as well as for historical analysis (Kübler et al. 2010). Paper 3 of this thesis proposes to use efficient operations on DNNF to introduce a polytime algorithm (once the DNNF is compiled) for enumeration of valid partial configurations.

Cluster trees

The cluster tree approach (Dechter and Pearl 1989) (also called join tree, junction tree, clique tree) relies on finding clusters of variables such that the interactions between the clusters are tree-structured, which allows solving the queries about a compiled problem by efficient tree algorithms. For example, cluster trees were proposed for use in product configuration at Renault (Pargamin 2002; Pargamin 2003). The same clustering idea underlines the tree decomposition used in DNNF compilation (Darwiche 1999; Darwiche 2004), which makes the result of cluster tree compilation similar to the result of DNNF compilation. Thus, we will not analyze the cluster tree approach separately from DNNF.

Automata and multi-valued decision diagrams

Automata were proposed for interactive configuration and explanation of invalid configurations by Amilhastre et al. (2002), with the compilation procedure based on the work of Vempaty (1992), who proposed to use automata to solve CSPs. Such automata closely resemble MDDs. Automata and MDDs have similar sizes of compiled representation (Hadzic and Hansen 2008), which are typically smaller than the sizes of BDDs for product configuration problems (Amilhastre et al. 2002; Hadzic and Hansen 2008). Amilhastre et al. (2002) report the size of the compiled automata representation to be 3.4 Mb vs 29.5 Mb for BDDs for the product configuration data of Renault Megane with 10^{12} valid configurations. Approximate compilation of MDDs (Hadzic, Hooker, et al. 2008) can give further space and time reductions while still providing useful data for interactive product configuration tasks. Given such properties, the methods appear to be promising, and might be a topic of future work.

Tree-driven automata

Tree-Driven Automata (Fargier and Vilarem 2004) can be compared to d-DNNF, but relaxing the requirement of only having Boolean variables. Although tree-driven automata have been proposed for product configuration (Fargier and Vilarem 2004), neither empirical evaluations nor implementations are available, which makes it difficult at the moment to estimate the benefits of this method.

AND/OR Decision Diagrams

Each node in an AND/OR BDD (Mateescu and Dechter 2006) represents a formula of the form: $(((a_1 \wedge a_2 \wedge \cdots \wedge a_n) \wedge x) \vee ((b_1 \wedge b_2 \wedge \cdots \wedge b_m) \wedge \neg x))$, where a_i and b_i are functions represented by the children of the node and x is the decision variable of the node. Hence, AND/OR BDDs define a subset of d-DNNF and satisfy the decomposable property. The AND/OR MDDs (Mateescu and Dechter 2006) are multi-valued versions of AND/OR BDDs. Compiled tree-driven automata (Fargier and Vilarem 2004) are essentially the same as (although developed independently from) AND/OR MDDs (Mateescu and Dechter 2006), with minor difference in the compilation approach, which is guided by a tree-decomposition for tree-driven automata, and by variable-elimination based algorithms for AND/OR MDDs, while variable elimination and cluster-tree decomposition are, in principle, the same (Dechter and Pearl 1989). AND/OR MDDs were proposed for use in product configuration (Mateescu, Dechter, and Marinescu 2008; Mateescu and Dechter 2008), but no empirical evaluation is available yet. Comparing performance of the tools for AND/OR MDD and d-DNNF compilation may be a topic of future work, especially since at least the binary code[1] of an AND/OR MDD compiler is available.

Tree-of-BDDs

Tree-of-BDDs is a data structure that was introduced to address the problem of huge BDDs that turn up in for many configuration problems (Subbarayan 2005). It uses tree decomposition of the constraint graph and compiles each individual BDD. The method relies on the total size of the partial BDDs being smaller than the size of the monolithic BDD. The compilation time for Tree-of-BDDs can be shorter than for d-DNNF (Subbarayan, Bordeaux, et al. 2007). However, Tree-of-BDDs supports less number of polytime operations. For example, the operation of checking the validity of an extra clause can not be performed on Tree-of-BDDs as efficiently as on sd-DNNF (Subbarayan, Bordeaux, et al. 2007; Fargier and Marquis 2009), which limits the applicability of Tree-of-BDDs for answering at least some of the configuration questions. Still, promising performance (Subbarayan, Bordeaux, et al. 2007) and available implementation[2] make Tree-of-BDDs an interesting topic for future work.

4.3 Solver benchmark

The previous sections introduced many methods that were used to tackle configuration problems. Often some tools implementing some methods are better on some problem instances, but not on all, as many comparisons show (Walsh 2000; Bennaceur 2004; Bordeaux et al. 2006; Hamadi and Bordeaux 2007; Pan and Vardi 2005; Mendonça 2009; Pohl et al. 2011). This section benchmarks several tools to find the most appropriate one to work with the product configuration data of our industrial partners in the research project. The tools were selected based on good performance in benchmarks

[1] `http://graphmod.ics.uci.edu/group/aomdd`

[2] `http://www.itu.dk/people/sathi/tob/`

within their specific disciplines, as well as on the implementation of well-known algorithms and availability of the source-code that could be investigated.

4.3.1 Testbed

For a comparison we use a truck configuration problem, we will call it Dataset A. We also use three car configuration problems from DaimlerChrysler AG, C210_FVF, C211_FW and C638_FKA (Küchlin and Sinz 2000; Sinz et al. 2003). A set of smaller problems was constructed based on Dataset A by reducing the number of constraints in it to see how the running time increases with the problem size. Dataset A has 53818 constraints and 511 variables; average domain size is 6.37. Details about the other three problems are given in Table 4.1.

The constraints in Dataset A are of two types. The first type of constraints represents forbidden combinations of values of the following form:

$$\neg\left((x_{k_1} = a_{k_1}) \wedge \ldots \wedge (x_{k_L} = a_{k_L})\right).$$

Each constraint of this type contains from two to ten values. The second type of constraints represents direct implications of the following form:

$$(\neg(x_k = a_k)) \vee (x_l = a_l),$$

Both of these two types of constraints can be directly converted to CNF clauses when using direct encoding (that is, encoding each value with one Boolean variable):

$$(\neg(x_{k_1} = a_{k_1}) \vee \ldots \vee \neg(x_{k_L} = a_{k_L})).$$

The problems C210_FVF, C211_FW, C638_FKA are in DIMACS CNF format[3], which means that all variables have Boolean domain, and each constraint is a disjunction of literals, where a literal is either a positive or negative atomic proposition.

4.3.2 Algorithms and tools

Six tools were used for the tests. The constraints were converted to a format suitable for each tool. Solvers and formats are illustrated in Figure 4.3 and described below.

[3] http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps

Table 4.1: Details for problems C210_FVF, C211_FW, C638_FKA.

Problem	Boolean Variables	CNF Clauses
C210_FVF	439	1853
C211_FW	327	3186
C638_FKA	528	5346

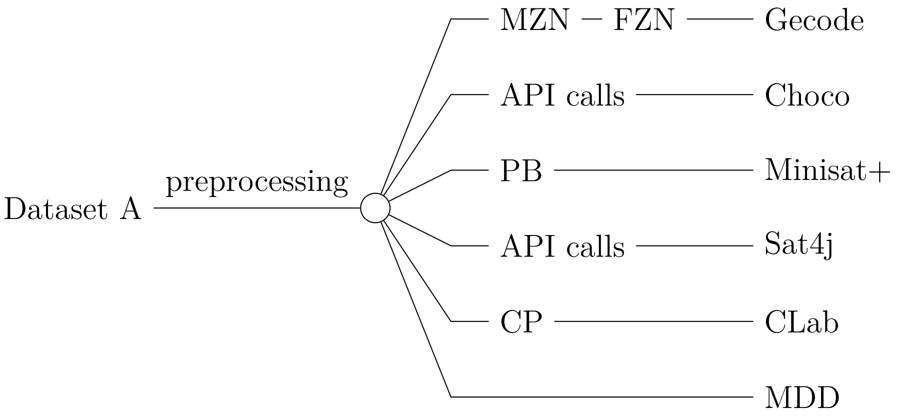


Figure 4.3: The tools and intermediate formats used. MZN, FZN – MiniZinc and FlatZinc file formats, PB – Pseudo Boolean file format, CP – CLab file format.

Two general-purpose constraint solvers were used, Gecode 3.2.2[4] and Choco 2.1.1[5]. Input data for Gecode was in MiniZinc/FlatZinc formats (Nethercote et al. 2007). MiniZinc files were created for all the problems, and then the MiniZinc files were converted to FlatZinc format using G12 MiniZinc-to-FlatZinc converter[6]. Input data of Choco was loaded via its API. Due to the direct use of the API and not a file format, all timing results for Choco exclude the time needed for reading the data from a disk, and include only the time needed to populate in-memory data structures of the solver and the time to execute the solving procedure.

As SAT-solvers, Sat4j 2.1.0 (Le Berre and Parrain 2010), Minisat+ and Minisat 2.0 (Een and Sörensson 2004; Een and Sörensson 2006) were used. The input data of Sat4j was loaded through the API, hence the disk reading time is not included in the timing results, similar to Choco. Minisat+ was used for Dataset A with input data in Pseudo-Boolean format[7] (Een and Sörensson 2006), and Minisat 2.0 was used for C210_FVF, C211_FW, C638_FKA with the data in DIMACS CNF format. The reason for using Minisat+ instead of the plain Minisat for Dataset A is that Minisat+ takes care of encoding finite-domain variables into Boolean variables.

As a BDD-based tool, CLab 1.0 (Jensen 2004b) was used with input data in the CLab file format (Jensen 2004a).

Another BDD/MDD-tool was implemented, as part of this research, in C++ using a BDD representation with logarithmic encoding of finite-domain variables. The C/C++ package BuDDy was used for this purpose (Lind-Nielsen 2002). The pre-ordering algorithms from (Narodytska and Walsh 2007) were implemented for sorting variables and constraints, using the inflation parameter $r = 1.5$ in the clustering step. A simplified version of the MCL clustering algorithm (Dongen 2000) was used, skipping the truncation heuristics and the sparse matrix multiplication tools. No post-ordering of the variables was included.

[4] http://www.gecode.org

[5] http://choco.sourceforge.net

[6] http://www.g12.csse.unimelb.edu.au/minizinc/

[7] For format description, see http://www.cril.univ-artois.fr/PB05/

Other knowledge compilation methods were not included in the benchmark due to lack of time. However, preliminary tests with d-DNNF compilers c2d (Darwiche 2004) and DSHARP (Muise et al. 2012) have shown that they can compile up to 80% of constraints of Dataset A, but both failed to compile the complete dataset due to the memory limit of 1 GB.

All tools were run with default settings and no extra tuning. It is possible to argue that it is not fair to take a solver and just use it with no tuning, especially since it was shown that automatic parameter tuning of a solver can significantly increase its performance (Hutter et al. 2009). However, the industry would like to have a trouble-free no-support tool that just works, especially since the sets of variables and constraints are changing continuously, and it might be prohibitively expensive to continuously re-tune or re-evaluate solvers.

The tools used for benchmarking were state-of-the-art at the time the experiments were conducted (2009), but at the moment of writing (2012) new tools and updates have become available. It might be a topic of future work to make new experiments with the newly available tools.

4.3.3 Benchmarking time to compile and get the first answer

In the first benchmark a solver was to answer whether there exists at least one valid configuration, and it is known in advance that there exists one for each problem (all instances are satisfiable). The main goal was to evaluate whether the compilation time for knowledge compilation methods is acceptable or not, as well as whether the search-based tools can handle the data.

The time limit for all the tools was 1 hour, and memory limit was 1 GB. All benchmark problems were executed on a 2.33 GHz Intel Core 2 Duo processor with 3.25 GB of RAM; only one core was used in the benchmark due to single-threaded implementations of the algorithms.

Results Timing results for Dataset A are presented in Figure 4.4. Timing results for problems C210_FVF, C211_FW, C638_FKA are presented in Figure 4.5. The results show that BDD and MDD based tools were not able to compile the complete Dataset A, while all search-based tools were able to successfully handle it. Runtime variation between search-based tools differed not significantly, and any of them would suit to answer whether there exist at least one valid product configuration. The results also show that Minisat+ performs worse than Minisat, with the most probable explanation being that the former solver performs a re-encoding.

Often we are interested in more intricate questions, and answering them with the help of search-based tools would require answering multiple (related) questions. Answering multiple related questions can significantly benefit from incremental solving, which is discussed in the next section.

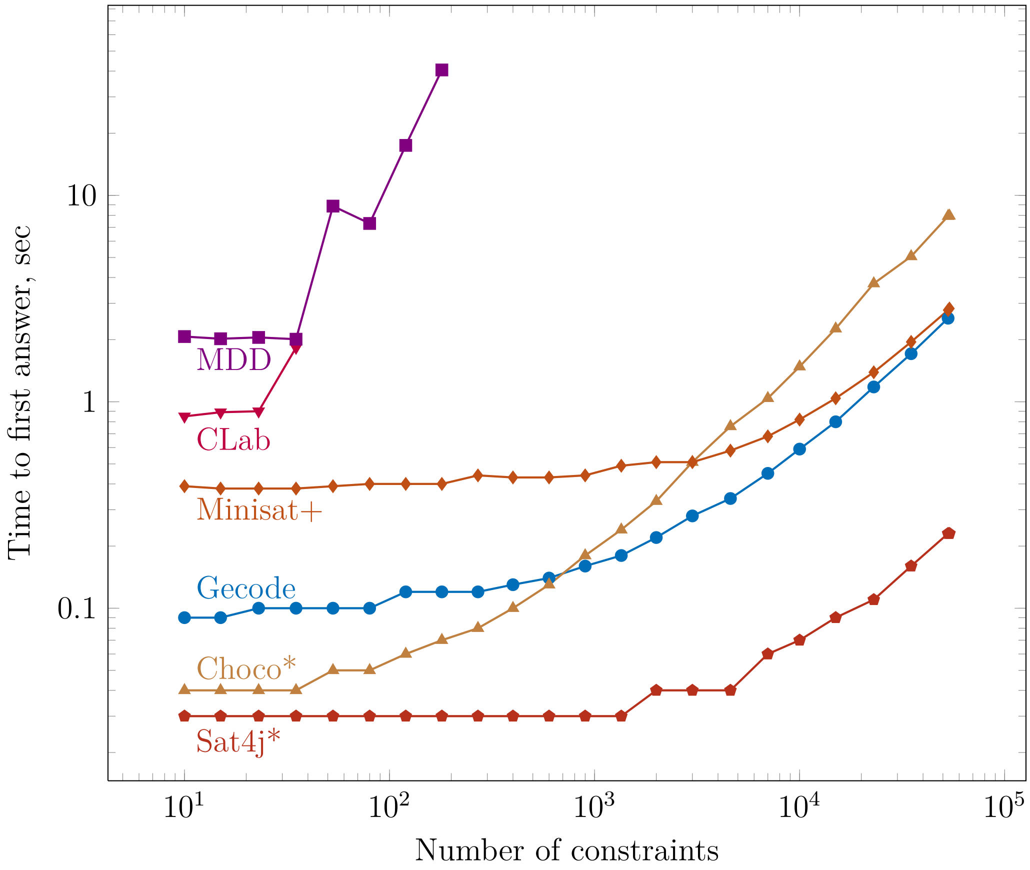


Figure 4.4: Time to get the first answer from each solver for Dataset A benchmark and its reduced versions.

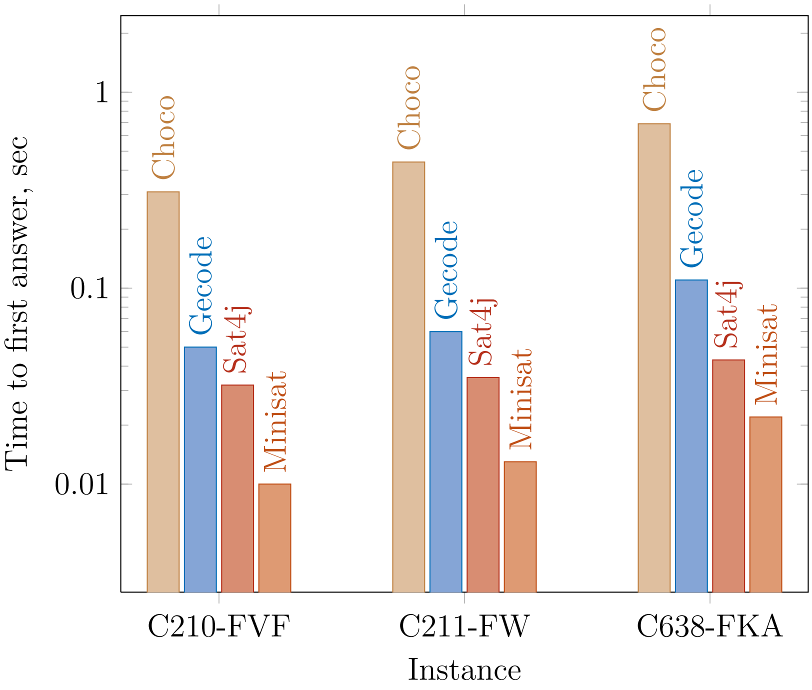


Figure 4.5: Time to get the first solution, DaimlerChrysler AG benchmarks.

4.3.4 Benchmarking time to get consecutive answers: incremental solving

Some problems can be solved by search-based solvers through iteratively solving a number of related instances. Traditional examples that require multiple SAT-solver queries are Bounded model checking (BMC) (Clarke, Biere, et al. 2001) and Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) (Bradley and Manna 2007; Bradley 2011). However, product configuration also has problems that can be solved by multiple queries to a solver, for example, enumeration of valid configurations (see Paper 3), verification of Item Usage Rules (see Paper 2), and interactive product configuration. Since solving multiple related instances is useful for product configuration, knowledge compilation methods including BDDs and MDDs seem very appealing. We wanted to evaluate whether search-based tools—especially the ones capable of incremental solving (Een and Sörensson 2003)—can come close to the performance of compiled data structures, and whether the average running time of the search-based tools is acceptable and how often the search-based tools show worst-case behavior.

Interactive configuration was chosen as a benchmark for measuring both the speed-up gained from incremental solving and the average run-time. In the interactive configuration a user selects values for the variables one-by-one, and the configurator must guarantee backtrack-freeness (no dead ends) and completeness (all valid configuration should be reachable by the user) of the process by restricting the user choices to the valid domains. We measured how long it takes to compute whether a value belongs to the valid domain of a variable. Since the configuration instance is the same, it allows reusing some inferred information between the queries.

The most widespread incremental interface of a SAT-solver is the incremental interface of Minisat proposed in (Een and Sörensson 2003), which allows forcing values for a set of variables without modifying the state of the model or data. Since user choices are exactly the assignments of values to variables, such incremental solving fits well for interactive configuration. To test whether a value belongs to the valid domain, the value is temporarily added to the user choices and forced in the solver. If there exists a satisfiable assignment, then the value belongs to the valid domain and the user is permitted to select the value. Otherwise, the value is outside of the valid domain of the variable, and the user is not allowed to select it. To test multiple values and variables, a simple simulator was implemented. It simulated user actions of choosing values for variables. Among all unassigned variables, a variable was selected randomly. The time to compute whether each value of the chosen variable belongs to the valid domain was measured. Then, one value from the valid domain was selected and fixed, and the process was repeated one level deeper. From several such simulations, the average and the maximum times were computed.

The CSP solvers considered (Choco and Gecode) had no incremental capabilities built-in. Implementing such functionality might be difficult, and even more difficult would be to maintain the codebase of a solver afterwards, since the main users of the solvers do not demand this feature, and it will be a burden for the core developer. Thus, CSP solvers were not considered in this benchmark. Incremental capability for

Gecode was implemented within a MSc thesis (Sachenkova and Thapaliya 2011), and experiments have shown that the speedup is less than the speedup achievable by a Minisat-like SAT-solver.

For MDDs, a simple test of validity of one value of one variable was taken, which was based on operations Restrict and Compute One Solution. Theoretical time bounds for MDDs predict that the run-time would be very similar between different variables and values, thus the simulator used for Sat4j was not used for MDDs.

Other knowledge compilation methods might be suitable for interactive configuration, but due to lack of time they were not included in the benchmark, and might be a topic of future work. Especially interesting is sd-DNNF with computed partial derivatives described in (Darwiche 2001b), as the use of derivatives allows to reason about changes only, without the need to reevaluate values for the complete data structure.

Results Experimental results are presented in Figure 4.6. The data shows that MDD was an order of magnitude faster than Sat4j. It also shows that there is a significant speed-up when using incremental capabilities of Sat4j compared to starting the solver "from scratch". The data also shows that the average runtime of the solver, as well as observed longest run-times, are acceptably short.

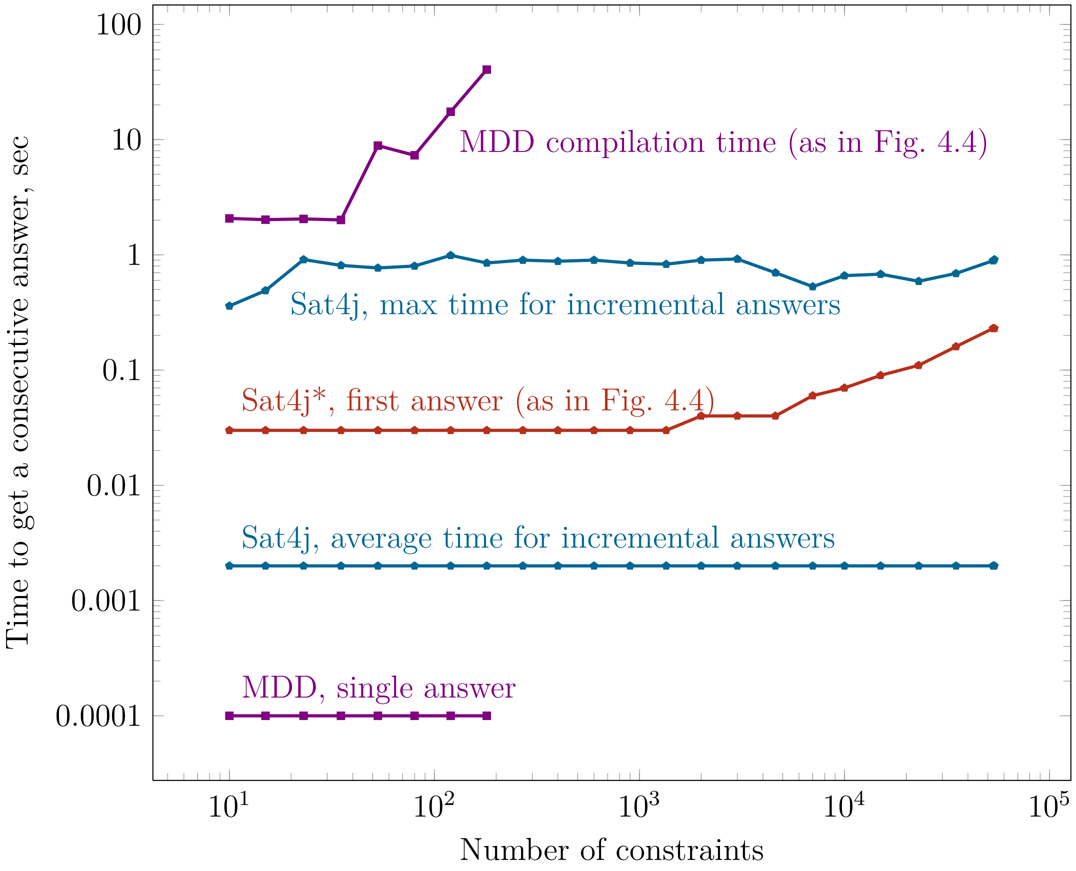


Figure 4.6: Incremental solving for Dataset A and its reduced versions.

4.4 Explaining efficiency

Our experiments have shown that SAT-solvers work very well on some industrial configuration problems (Voronov, Åkesson, and Ekstedt 2011; Voronov, Åkesson, Tidstam, et al. 2012), and not so well on, for example, supervisory control problems (Voronov and Åkesson 2008). Since both mentioned problems are similar in the number of variables and the number of constraints, the difference in SAT-solver performance must lie not in the size of the problems, but perhaps in differences in the problem structures. This section looks at *Parameterized Complexity* (Downey and Fellows 1999), which attempts to characterize problems in more refined ways than purely by the problem size as in traditional complexity analysis, allowing to better explain and predict the difference between the problems.

In classical complexity theory the computational complexity of a problem is considered exclusively in terms of the *input size*, and structural properties of problem instances are not represented. The framework of *Parameterized Complexity* addresses this issue (Downey and Fellows 1999; Flum and Grohe 2006; Niedermeier 2006). Parameterized complexity tries to find a parameter $k = \pi(F)$ of instances F that is smaller than the size n of the instance, such that there is an algorithm with running time polynomial in n, and exponential only in k, that is, $2^k \cdot n^{\mathcal{O}(1)}$. Such algorithms are called *fixed parameter tractable* (fpt) algorithms.

As an example, a typical problem in this thesis has thousands of variables and tens of thousands of constraints. The best known worst-case run-time bound for 3-SAT is $\mathcal{O}(1.321^n)$, where n is the number of variables (Hertli et al. 2011). For $n = 1000$ such a bound would suggest runtimes of order of at least 10^{100} seconds (for comparison, the age of the universe is estimated to be about 10^{17} seconds). However, the observed practical runtimes are less than a second. A possibility to explain such short runtimes would be a fpt-algorithm with polynomial runtime in n, and exponential in some fixed parameter $k \ll n$.

One important notion related to fpt-algorithms is a *backdoor* (Williams et al. 2003). A *backdoor set* is a set of variables such that when instantiated, the problem becomes “easy”, that is it simplifies to some tractable class. There is a number of such tractable classes to which original problems can be simplified, for example, Horn formulas, which have at most one positive literal in each clause, Renameable Horn formulas (RHorn), which have a variable renaming that makes a formula into a Horn formula, or 2-SAT, which have at most two literals per clause. Horn and 2-SAT are linear-time solvable (Dowling and Gallier 1984; Aspvall and Plass 1979), and RHorn is polynomial-time solvable (Lewis 1978). An algorithm that efficiently solves the simplified problem is called a *sub-solver.* A *strong* backdoor set is one for which the sub-solver can find a satisfying assignment or decide unsatisfiability for *any* instantiation of the variables in the backdoor set. A *weak* backdoor set is one for which the sub-solver can provide a satisfying assignment for *at least one* instantiation (finding such instantiation is still hard). Weak backdoors are not relevant for unsatisfiable instances, which have no satisfying assignments, while strong backdoors are relevant for both satisfiable and unsatisfiable instances. Since an instance becomes efficiently-solvable by a sub-solver after any instantiation of the variables in a strong backdoor set, the instance can be considered fixed parameter tractable with respect to the size of the backdoor set. Experiments with identification of backdoors for automotive problems were performed in (Dilkina et al. 2007; Samer and Szeider 2008; Li and Beek 2011).

Another important and widely-studied property of SAT instances is *treewidth* (Robertson and Seymour 1984), which informally can be described as a “tree-likeness” of a constraints graph. The most prominent graph representation of a CNF formula F is the *primal graph* $G(F)$. The vertices of $G(F)$ are the variables of F; two variables x, y are joined by an edge if they occur in the same clause, that is, if $x, y \in \mathrm{var}(C)$ for some $C \in F$. A *tree decomposition* of a graph $G = (V, E)$ is a tree $T = (V', E')$ together with a labeling function $\chi : V' \to 2^V$ associating to each tree node $t \in V'$ a bag $\chi(t)$ of vertices in V such that the following tree conditions hold:

1) every vertex in V occurs in some bag $\chi(t)$;

2) for every edge $xy \in E$ there is a bag $\chi(t)$ that contains both x and y;

3) if $\chi(t_1)$ and $\chi(t_2)$ both contain x, then each bag $\chi(t_3)$ contains x if t_3 lies on the unique path from t_1 to t_2.

An example of a graph and its tree decomposition is shown in Figure 4.7.

The *width* of a tree decomposition is $\max_{t \in V'} |\chi(t)| - 1$. The *treewidth* of a graph is the minimum width over all its tree decompositions. The treewidth of a graph is a measure for its acyclicity, i.e., the smaller the treewidth the less cyclic the graph is. In particular, a graph is acyclic if and only if it has treewidth 1.

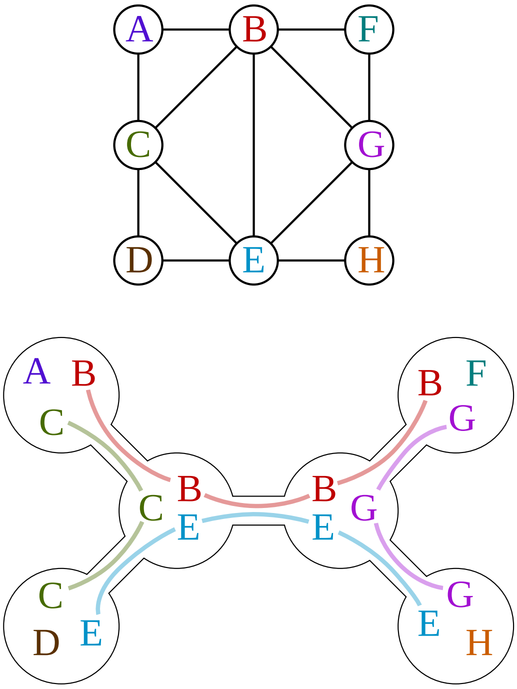


Figure 4.7: Sample graph and its tree decomposition. Adapted from (Eppstein 2007).

If its tree decomposition is given, an instance can be solved using a bottom-up dynamic programming approach on the tree decomposition, making the problem fixed-parameter tractable for instances with bounded treewidth (Gottlob et al. 2002).

The next subsection investigates the backdoor sizes and the tree-width of two industrial instances.

4.4.1 Evaluating industrial product configuration instances

Sizes of RHorn backdoors (Kottler et al. 2008; Dilkina et al. 2007) and tree-width were computed for two industrial product configuration instances. The results are shown in Table 4.2. As seen from the table, we could not find a parameter less than 75, which would suggest that the worst-case running time should be in the order of 2^{75} operations; so that if a modern computer performs about 10^9 operations per second, the running time would be more than 10^{13} seconds, which is much more than the observed running times of less than one second. To achieve one second as an upper bound, a parameter $k \approx 30$ would be needed. Thus, neither RHorn backdoor sizes nor tree-width of the instances can explain the good SAT-solver performance on these instances.

Tree-width was earlier studied as a candidate for explaining a good run-time behavior of SAT-solvers on industrial instances of the 2009 SAT competition (Mateescu 2011). It was found that many instances that were solved fast in practice have rather large tree-width (Mateescu 2011), which supports our observation that tree-width might not be suitable for explaining good performance of SAT-solvers.

Since none of the parameters we considered can explain the good experimental performance of SAT-solvers, there is a need for better problem characterization methods.

Table 4.2: Backdoor sizes and tree-width bounds for two industrial product configuration problems. Ladder and quadratic encodings of finite-domain variables are discussed in Section 3.1.3.

Property	**Problem A**		**Problem B**	
	Encoding		Encoding	
	Ladder	Quadratic	Ladder	Quadratic
General properties				
CNF variables	5901	3206	1907	1596
CNF clauses	65183	91644	9010	11270
Max clause length	108	108	60	60
Avg clause length	2.5	2.3	2.5	2.4
FPT properties				
Tree-width upper bound	488	1579	414	432
Tree-width lower bound	136	135	76	75
RHorn backdoor set size upper bound	1255	2243	398	433
RHorn backdoor set size lower bound	1255	1254	187	185

4.5 Conclusions

We evaluated several knowledge compilation tools and several search based tools for use with product configuration data from our industrial collaborator. The results show that MDD fails to compile the complete dataset within the time and memory limits, while search-based tools can successfully handle the complete dataset. SAT-solvers with incremental interface appear to work extremely well for the data we used. Other knowledge compilation methods require further investigations. We also investigated the reasons for good performance of some of the tools, and have not found yet a definitive explanation.

Chapter 5

Using Supervisory Control Theory for Interactive Product Configuration

Compiling configuration constraints into a representation that allows efficient reasoning—a technique called *knowledge compilation* introduced in Section 4.2—can save significant amounts of time when answering multiple queries on the same configuration problem. Examples of multiple queries on the same data include enumeration of valid partial configurations, interactive configuration, or in situations when multiple users use the same data. However, traditional knowledge compilation methods—like Binary Decision Diagrams (BDDs) introduced in Section 4.2.1—usually have high memory requirements, which might be a limiting factor for using compiled data structures on low-memory devices. Another application that requires small sizes of compiled data structures is a client-server architecture, where a server sends all the data necessary for configuration to a client over a limited-bandwidth channel. Applications like these motivated, for example, a technique for BDD compression (Hansen and Tiedemann 2007), a technique for compression of Multivalued Decision Diagrams (Hadzic, Hansen, and O'Sullivan 2008), and a new knowledge compilation technique—Tree-of-BDDs—based on a decomposition of the data (Subbarayan 2005). However, even more compact representations of the configuration data are desirable, and this might potentially be achievable by using a different approach – Supervisory Control Theory (SCT) introduced in Section 3.2. To the best of our knowledge, SCT, and particularly SCT synthesis, has not been used for product configuration problems before.

The product configuration tasks that are approached by SCT in this chapter are *forward-only interactive configuration*, *interactive configuration with undo-actions*, and *reconfiguration*, all introduced in Section 2.3. Moreover, efficient interactive configuration can be used to efficiently enumerate valid partial configurations too, as explained in Section 2.3.

We propose to use SCT for product configuration in an approach that consists of the following steps:

1) encode variables (families) and constraints, as well as additional specifications when needed, as automata;

2) synthesize a supervisor;

3) use the supervisor in combination with the original automata to efficiently answer the interactive configuration or reconfiguration queries.

Automata have been previously used for solving Constraint Satisfaction Problems (CSP) (Vempaty 1992), as well as product configuration problems (Amilhastre et al. 2002). However, in these approaches a CSP is represented by a monolithic automaton resulting from applying product and minimization algorithms (Hopcroft and Ullman 1979; Kimura and Clarke 1990), which might have size exponential in the size of the original CSP. Our approach does not require to monolithically compose all automata. Instead, our approach relies on supervisory synthesis algorithms to do the necessary computations. Synthesis algorithms, in turn, can either do a brute-force monolithic synchronization of all automata, or use more efficient strategies, including compositional synthesis that avoids synchronizing all automata (Mohajerani et al. 2011), or symbolic synthesis that uses BDDs to improve efficiency (Hoffmann and Wong-Toi 1992; Vahidi et al. 2006). Moreover, SCT allows representing a supervisor not only as a monolithic automaton, but also as a set of automata (Mohajerani et al. 2011), or as logic expressions (Miremadi et al. 2011), which might potentially require less memory than other representations.

5.1 Encoding interactive product configuration using supervisory control theory

Different capabilities of interactive configuration require different levels of sophistication of encodings. We will start with the simpler forward-only interactive configuration and will add undo-actions and reconfiguration afterwards. All encodings create an automaton for each variable and each constraint, these are treated as plants. Undo-actions and reconfiguration require an extra automaton, which is treated as a specification. Specification for the forward-only interactive configuration does not have separate automata, and is expressed as marked states only.

The overall algorithm is illustrated in Algorithm 5.1. The algorithm converts all constraints to Conjunctive Normal Form (CNF) to simplify the encoding; such conversion is described in Section 3.1.3. The algorithm encodes all variables and constraints as automata; the encodings are different between forward-only configuration and configuration with undo-actions. For configuration with undo-actions and reconfiguration, the algorithm encodes extra specifications to ensure backtrack-freeness and correct reconfiguration. Plants and specifications automata are designated from previously created automata. Then, a non-blocking and controllable supervisor is synthesized; this synthesis procedure is denoted as NBC. The synthesized supervisor in conjunction with the original automata can be used for answering appropriate product configuration queries. The following sections describe the encodings in details.

Algorithm 5.1 Supervisor-for-Configuration-using-SCT

```
input: A CSP $\mathcal{P} = \langle X, D, C \rangle$ with finite-domain variables
  and propositional constraints
output: A supervisor for forward-only configuration,
  configuration with undo-actions, or reconfiguration
$C'$ = Convert-constraints-to-CNF($C$)
for $x \in X$:
  Encode-Variable-as-Automaton($x$)
for $c \in C'$:
  Encode-Clause-as-Automaton($c$)
Encode extra specifications when necessary
Define plants and specifications from created automata
Sup ← NBC(Plants, Specifications)
return Sup
```

5.1.1 Encoding forward-only interactive configuration

An automata encoding of a CNF formula F (which can be generalized to finite-domain variables as well) is as following. For each variable $x_i \in \mathit{scope}(F)$ introduce an automaton A_{x_i} with two states: an "unassigned" (initial) state $s^u_{x_i}$ and an "assigned" state $s^a_{x_i}$. Let the only marked state be $s^a_{x_i}$, since the product is not fully configured until every variable have a value assigned. Two transitions will lead from the unassigned state to the assigned state: one transition labeled x^T_i and the other transition labeled x^F_i, corresponding to assignment of the Boolean value True and False to the variable, respectively. The alphabet of the automaton is $\{x^T_i, x^F_i\}$. Formally, the automaton for the variable x_i is as follows:

$$A_{x_i} = \left\langle Q_{x_i}, \Sigma_{x_i}, f_{x_i}, q^0_{x_i}, Q^m_{x_i} \right\rangle,$$

where

$$\begin{aligned}
Q_{x_i} &= \{s^u_{x_i}, s^a_{x_i}\} \text{ is the set of states,}\\
\Sigma_{x_i} &= \{x^T_i, x^F_i\} \text{ is the alphabet,}\\
f_{x_i} &= s^u_{x_i} \xrightarrow{\Sigma_{x_i}} s^a_{x_i} \text{ is the transition function,}\\
q^0_{x_i} &= s^u_{x_i} \text{ is the initial state, and}\\
Q^m_{x_i} &= \{s^a_{x_i}\} \text{ is the set of marked states.}
\end{aligned}$$

For each clause $c_j \in F$, introduce an automaton, again with two states: the initial state where the clause is unsatisfied $s^u_{c_j}$, and another state $s^s_{c_j}$ where the clause is satisfied. The satisfied state will be the only marked state. For each literal in the clause, introduce a transition from the unsatisfied state to the satisfied one, labeled by the appropriate assignment of values (x^T_k for positive literal and x^F_k for negative literal). To allow multiple literals to be satisfied within the same clause, the satisfied state $s^s_{c_j}$ will have self-loops transitions for all of the literals. The alphabet of the automaton for the clause c_j will be $\{x^T_k \mid x_k \in c_j\} \cup \{x^F_k \mid (\neg x_k) \in c_j\}$. Formally,

$$A_{c_j} = \left\langle Q_{c_j}, \Sigma_{c_j}, f_{c_j}, q^0_{c_j}, Q^m_{c_j} \right\rangle,$$

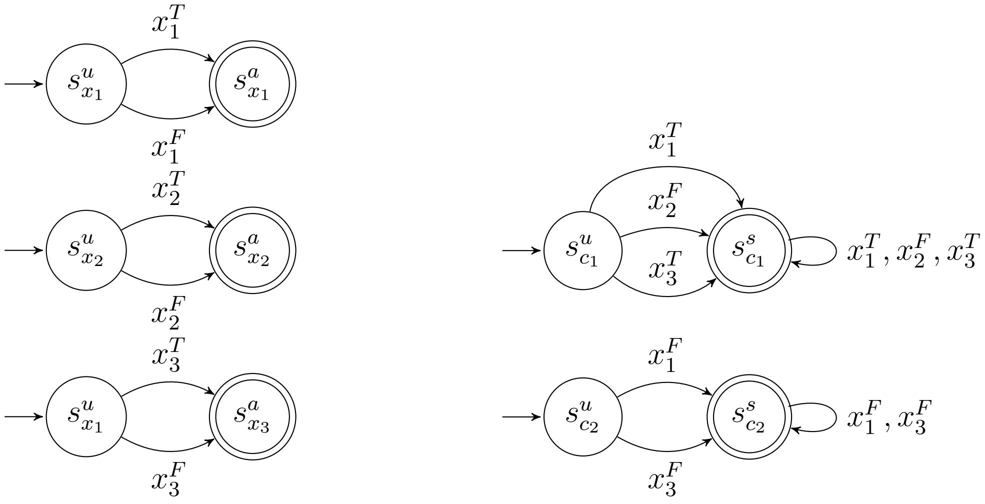
where

$$\begin{aligned}
Q_{c_j} &= \{s^u_{c_j}, s^s_{c_j}\} \text{ is the set of states,}\\
\Sigma_{c_j} &= \{x^T_k \mid x_k \in c_j\} \cup \{x^F_k \mid (\neg x_k) \in c_j\} \text{ is the alphabet,}\\
f_{c_j} &= \{s^u_{c_j} \xrightarrow{\Sigma_{c_j}} s^a_{c_j}\} \cup \{s^s_{c_j} \xrightarrow{\Sigma_{c_j}} s^s_{c_j}\} \text{ is the transition function,}\\
q^0_{c_j} &= s^u_{c_j} \text{ is the initial state, and}\\
Q^m_{c_j} &= \{s^s_{c_j}\} \text{ is the set of marked states.}
\end{aligned}$$

The complete model is a synchronous composition of automata for variables and clauses. However, a supervisor can be synthesized without explicitly constructing the automaton for synchronous composition. Automata for variables and clauses can be both considered as plants, while specification is expressed as marked states. However, since there are no uncontrollable events in this model, the distinction between plant automata and specification automata is not important for this model. A non-blocking supervisor for the complete model will ensure that taking any enabled events will lead to the marked state, which corresponds to the situation when all variables are assigned and all clauses are satisfied.

As an example, consider CNF $F = (x_1 \vee -x_2 \vee x_3) \wedge (-x_1 \vee -x_3)$. The variables x_1, x_2 and x_3 can be encoded into automata as shown in Figure 5.1a, and the two clauses can be encoded as shown in Figure 5.1b. The synthesized supervisor for the model is shown in Figure 5.2, where supervisor is shown with solid lines, while solid plus dashed lines represent the synchronous composition of variables and clauses automata (before synthesis).

The supervisor resulting from the synthesis has a lot of symmetry, since the order in which the variables are assigned is not fixed, resulting in the number of paths leading to the marked states being exponential in the number of variables. Restricting the order of variables will reduce the size of the supervisor, but will also reduce the flexibility of user choices. An automaton for restricting the order in which the variables are assigned to x_1, x_2, x_3 is shown in Figure 5.3. The supervisor for this order is shown in Figure 5.4.



(a) Automata for variables x_1, x_2 and x_3. The alphabets, top to bottom: $\{x_1^T, x_1^F\}$, $\{x_2^T, x_2^F\}$, $\{x_3^T, x_3^F\}$.

(b) Automata for constraints. Alphabets are: $\{x_1^T, x_2^F, x_3^T\}$, $\{x_1^F, x_3^F\}$.

Figure 5.1: Encoding CNF $F = (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor \neg x_3)$ as automata.

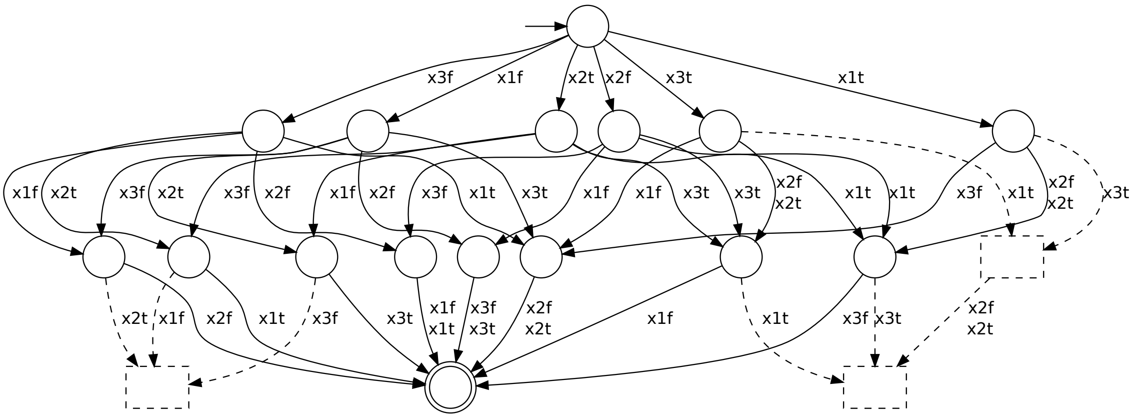


Figure 5.2: The automaton illustrating the supervisor (solid lines) and the states and transitions reachable by the unsupervised plant, but disabled by the supervisor (dashed lines).

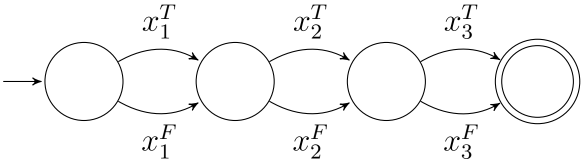


Figure 5.3: A specification to restrict the order of assigning the values to the variables to x_1, x_2, x_3.

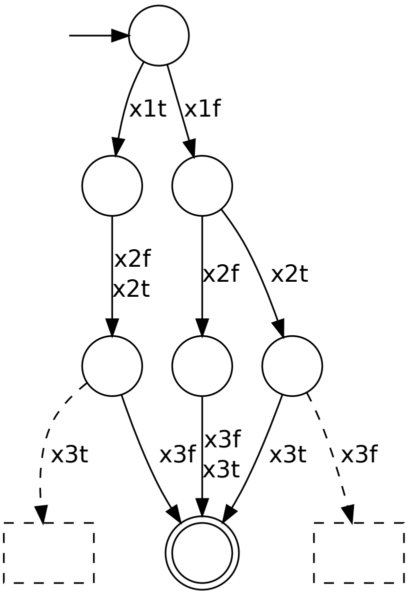


Figure 5.4: The automaton illustrating the supervisor (solid lines) assuming a fixed order of assignment (x_1, x_2, x_3) of the variables. Dashed nodes and transitions are reachable by the unsupervised system with the fixed order of variables, but are disabled by the supervisor.

5.1.2 Encoding interactive configuration with undo-actions

To illustrate undo-actions and reconfiguration, as well as to introduce by example an encoding for finite-domain variables, we will use a hypothetical car configuration example (similar to the example in Table 2.1, but slightly modified). The variables and their domains are shown in Table 5.1, and constraints are shown in Table 5.2.

To support undo-actions, variables are encoded by automata with $n + 1$ states, where n is the domain size of the variable, see Figure 5.5 for the encoding of the variable *Body*. One state, initial, is not marked and corresponds to the variable being unassigned. The other n states correspond to the values the variable can be assigned to (we will call these states *assigned* states). These assigned states are marked. The set of "assign" events is the same as in forward-only case, one event for one value of the variable. Each such event leads from the initial state to the corresponding assigned state. Each assign-event has a corresponding *undo*-event that leads back from the assigned state to the initial (unassigned) state. Due to the multi-valued nature of the variables, the negation of a variable is no longer encoded directly, and has to be represented in the constraints as a disjunction of all other values of the domain of the variable. For example, $\neg(\mathit{Body}{=}\mathit{Mini})$ have to be encoded as $(\mathit{Body}{=}\mathit{Sedan} \lor \mathit{Body}{=}\mathit{Suv})$.

Encoding of constraints requires three additions compared to the Boolean forward-only case. First, all negative literals have to be converted to disjunctions of positive literals. Second, due to undo-actions, encoding of a clause requires a separate state for each possible number of satisfied literal in the clause. However, since all literals are positive, and no two literals representing values of the same variable can be satisfied simultaneously, the number of states is limited by the number of variables, and not the number of literals. The number of states in an automaton for a clause with the scope of k variables will be $k + 1$. Assign-events for each literal lead from a state to the state corresponding to one more satisfied literal, for example, from 0 satisfied literals (unsatisfied clause) to 1 satisfied literal, from 1 to 2, etc. Thirdly, undo-events lead from a state to the previous one (for example, from 2 to 1, etc). Complete encoding of the constraint $\neg(\mathit{Body}{=}\mathit{Mini} \land \mathit{Engine}{=}\mathit{Gasoline})$, which can be rewritten as $(\mathit{Body}{=}\mathit{Sedan} \lor \mathit{Body}{=}\mathit{Suv} \lor \mathit{Engine}{=}\mathit{Diesel} \lor \mathit{Engine}{=}\mathit{Electric})$, is shown in Figure 5.6.

The specification needed to ensure the correct behavior for user undo actions is shown in Figure 5.7. This specification introduces two events *user_assign, user_unassign* that correspond to the user choices. The user can choose one of the two options:

- continue with the configuration by selecting event *user_assign*;

Table 5.1: Variables and their domains for the car configuration example.

Variable	Values
body	mini, sedan, suv
engine	gasoline, diesel, electric
transmission	manual, automatic, evt

Table 5.2: Constraints for the car configuration example.

$\neg((body = mini) \wedge (engine = gasoline))$
$\neg((body = mini) \wedge (engine = diesel))$
$\neg((body = sedan) \wedge (engine = electric))$
$\neg((body = suv) \wedge (engine = gasoline))$
$(engine = electric) \rightarrow (transmission = evt)$
$(transmission = evt) \rightarrow (engine = electric)$

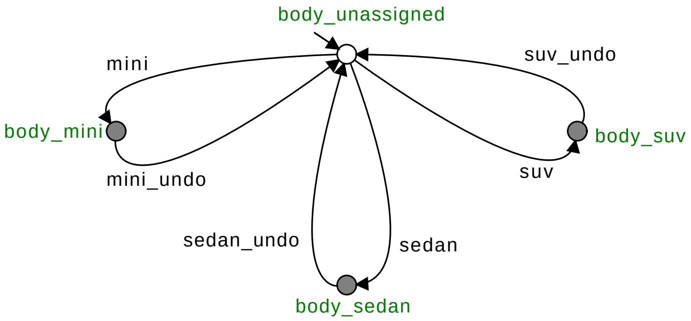


Figure 5.5: Automaton encoding variable *Body* with values *Mini*, *Sedan* and *Suv*.

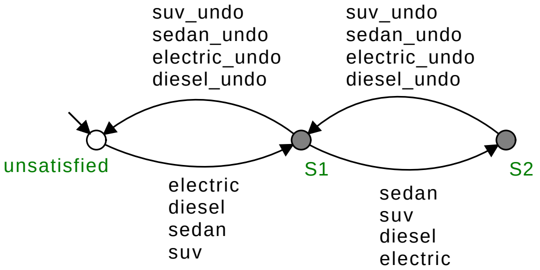


Figure 5.6: Automaton encoding constraint $\neg(\,Body{=}Mini \wedge Engine{=}Gasoline\,)$.

- undo some previous assignment by selecting event *user_unassign*.

Event *user_assign* is uncontrollable, making sure that the supervisor will always avoid dead ends and will be prepared to move "forward" with configuration, without relying on the user undo actions (backtracking) to get out of dead ends. Event *user_unassign* is controllable, since it must be disabled when no assignments have been made; making *user_unassign* uncontrollable would result in the specification being uncontrollable with respect to the plant, leading to a null supervisor. Consider automaton in Figure 5.7. Firing *user_unassign* leads to the state from where only undo-events are possible. However, if there are no events to undo, as is the case in the beginning of the configuration process, or after all assignments were already undone, the system will end up in a deadlock. To remove this deadlock, the supervisor have to disable the event *user_unassign*, thus this event have to be controllable.

The supervisor contains 101 states and 197 transitions, and it is illustrated in Figure 5.8. To give an intuition of interactive configuration session without relying on a drawing a supervisor, consider the following paths from the initial state to a marked state, where each row indicates the enabled events, and "∗" indicates the event taken:

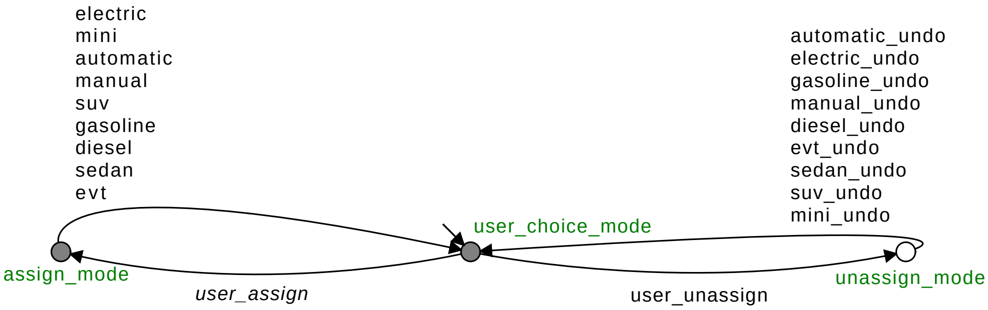


Figure 5.7: Automaton encoding backtrack-freeness for undo-actions.

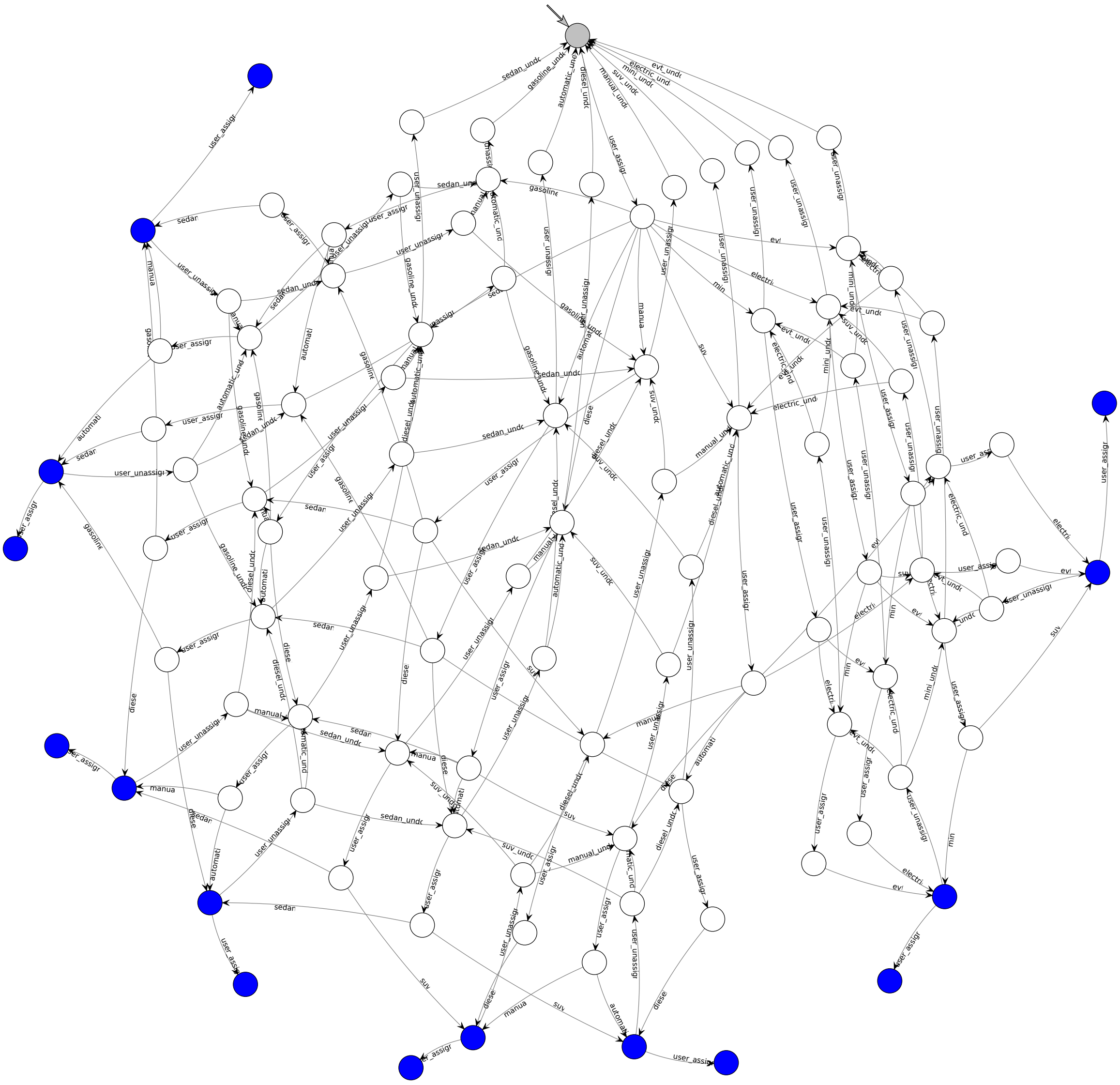


Figure 5.8: Illustration of the supervisor for interactive configuration with undo-actions for the car configuration example.

```
*user_assign
*mini, sedan, suv, gasoline, diesel, electric, manual, automatic, evt
*user_assign, user_unassign
electric, *evt
*user_assign, user_unassign
*electric
(done)

*user_assign
*mini, sedan, suv, gasoline, diesel, electric, manual, automatic, evt
*user_assign, user_unassign
electric, *evt
user_assign, *user_unassign       % example of undo-actions
*mini_undo, evt_undo
*user_assign, user_unassign
mini, suv, *electric
*user_assign, user_unassign
mini, *suv
(done)
```

The paths illustrate the choices offered to and made by the user. The intuition behind the supervisor is as following: after the user chooses *user_assign*, the supervisor disables all assignments that can not result in a valid complete configuration. The user is only offered the choices that will lead to a valid complete configuration (marked state) without the need for any undo-actions.

5.1.3 Encoding interactive configuration with undo-actions and reconfiguration

The reconfiguration procedure will be built on top of the solution for undo-actions. The specification needed to ensure the correct behavior for user undo actions and reconfiguration introduces three extra event, *user_force*, *user_force_start_undo* and *user_force_done* to the previously introduced events *user_assign* and *user_unassign*. This specification is shown in Figure 5.9. The *user_force* event can be either controllable or uncontrollable, the synthesis procedure results in the same supervisor. We chose to make it uncontrollable, to emphasize that the system should always be prepared for the user executing this action. Event *user_force_start_undo* is also uncontrollable. Event *user_force_done* is controllable, to ensure that the user has undone enough assignments to make partial configuration valid. With such specification, there are three options to choose from, when selected variants correspond to a valid partial configuration:

- continue with the configuration by selecting event *user_assign*;
- undo some previous assignment by selecting event *user_unassign*;
- enter a reconfiguration mode by selecting event *user_force*.

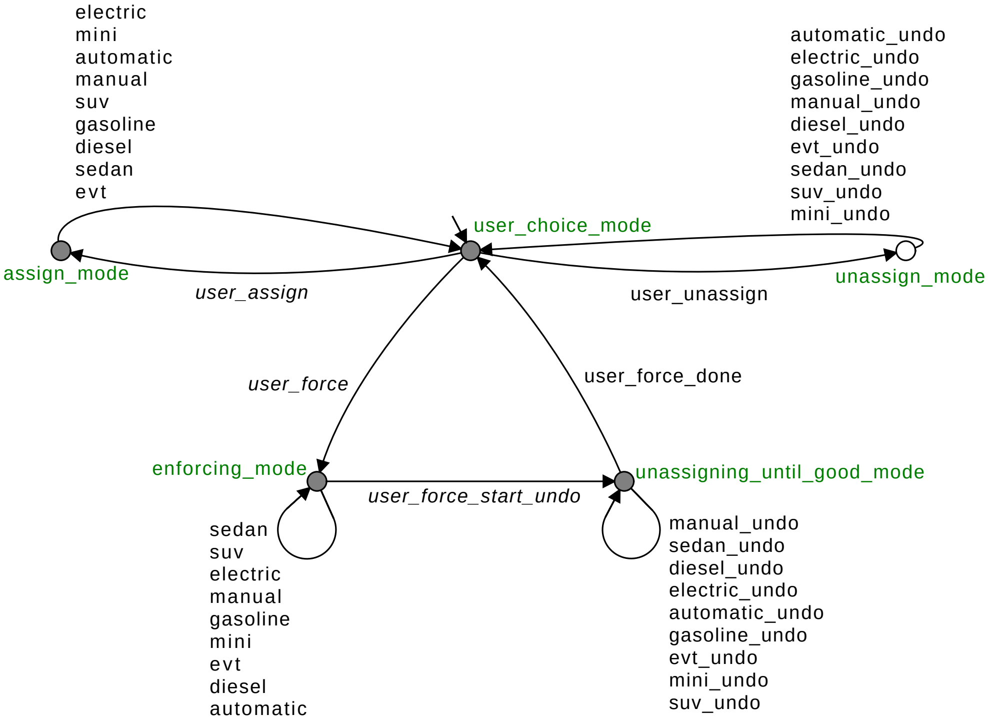


Figure 5.9: Automaton encoding user undo action and reconfiguration.

Naturally, these choices can be hidden from the user interface, presenting the user only with the variants (click once to assign a variant, click the variant second time to undo the assignment), firing the necessary events in the background.

The supervisor for undo actions and reconfiguration is illustrated in Figure 5.10. Unfortunately, with 229 states and 617 transitions, it is too big to be visualized properly on the standard printed page. The following path from the initial state to a marked state might give some intuition of the supervisor:

```
*user_assign, user_force
*mini, sedan, suv, gasoline, diesel, electric, manual, automatic, evt
*user_assign, user_unassign, user_force
electric, *evt
user_assign, *user_unassign, user_force    % example of undo-actions
*mini_undo, evt_undo
mini, *suv, electric
user_assign, user_unassign, *user_force    % example of reconfiguration
gasoline, *diesel, electric, user_force_start_undo
gasoline, electric, *user_force_start_undo
diesel_undo, *evt_undo, suv_undo
diesel_undo, suv_undo, *user_force_done
*user_assign, user_unassign, user_force
*automatic, manual
(done)
```

However, in this example even when the user chooses to force *Engine=Diesel*, there is an immediate offer to undo this choice. To make sure that the system offers to undo

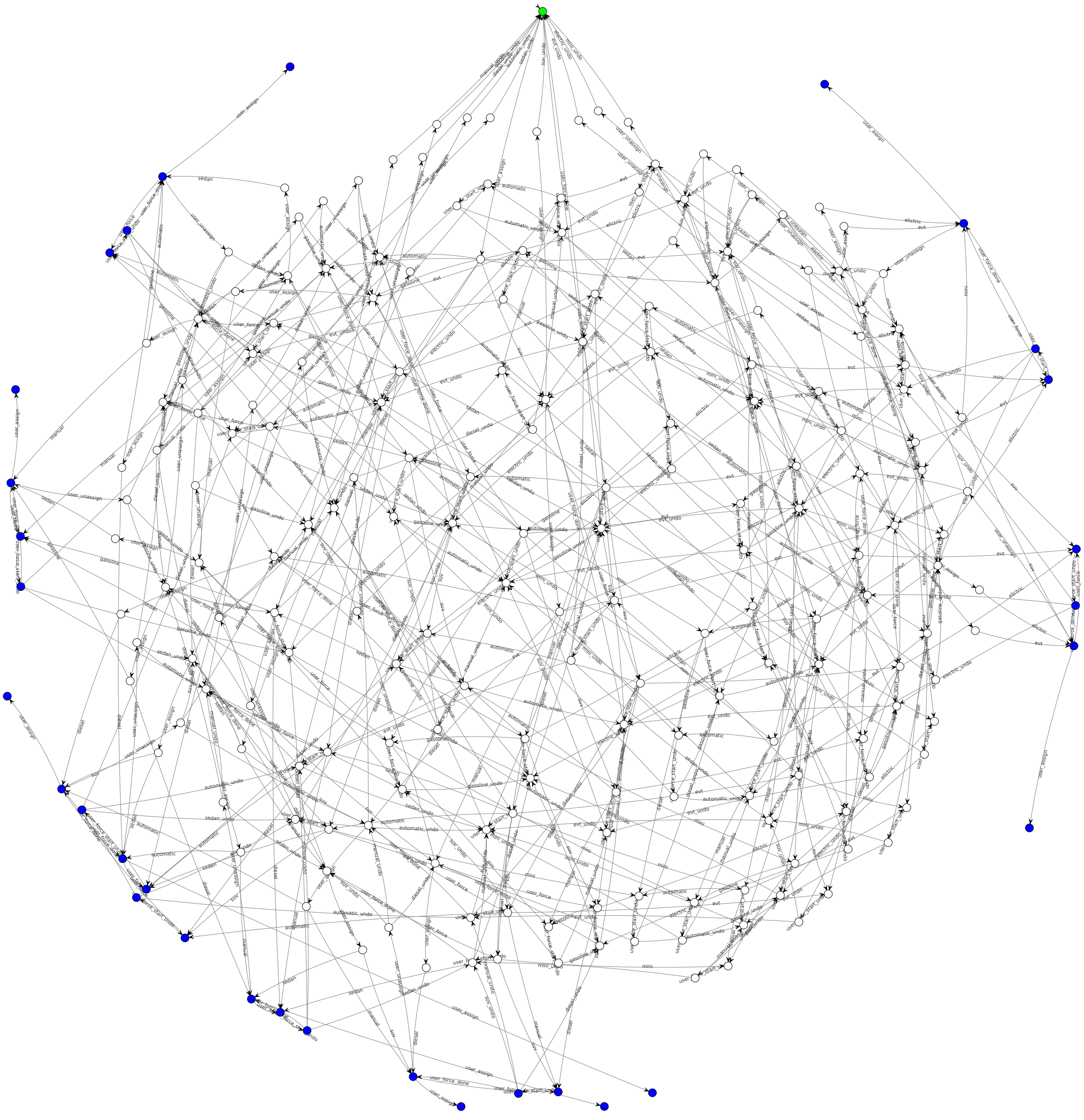


Figure 5.10: Illustration of the supervisor with undo-actions and reconfiguration for car configuration example.

only the choices that were not forced, the automata for the variables can be modified as shown in Figure 5.11. After event *user_force*, the automaton allows assigning values to the variable, but does not allow to unassign them until reconfiguration is over, that is, until the event *user_force_done* is fired. The supervisor for such automata contains 682 states and 1359 transitions.

This solution allows the user to undo actions and helps the user reconfiguring invalid configurations.

5.2 Representing the supervisor

The supervisor in Figure 5.4 had to disable only two transitions that led to blocking states. Instead of representing the supervisor as an automaton with unwanted transitions disabled, it is possible to augment existing automata with conditions that will specify when the transition is allowed. Such conditions are called *guards*. The supervisor represented by such guards can potentially be smaller than, for example, the BDD for representing the data necessary for the configuration process. The algorithm for computing such guards was introduced in (Miremadi et al. 2011). The generated guards might be compared to extra constraints added by, for example, Adaptive-Consistency algorithm (Dechter and Pearl 1987) for ensuring backtrack-freeness of the CSP search.

The supervisor in Figure 5.4 can be represented by two symbolic guards shown in Table 5.3, one guard for event x_3^T, which would allow x_3^T to be fired only when the automaton for clause c_2 is in the state $s^s_{c_2}$, that is, clause c_2 is already satisfied, and one guard for event x_3^F, allowing it to be fired only when in the state $s^s_{c_1}$, that is, when

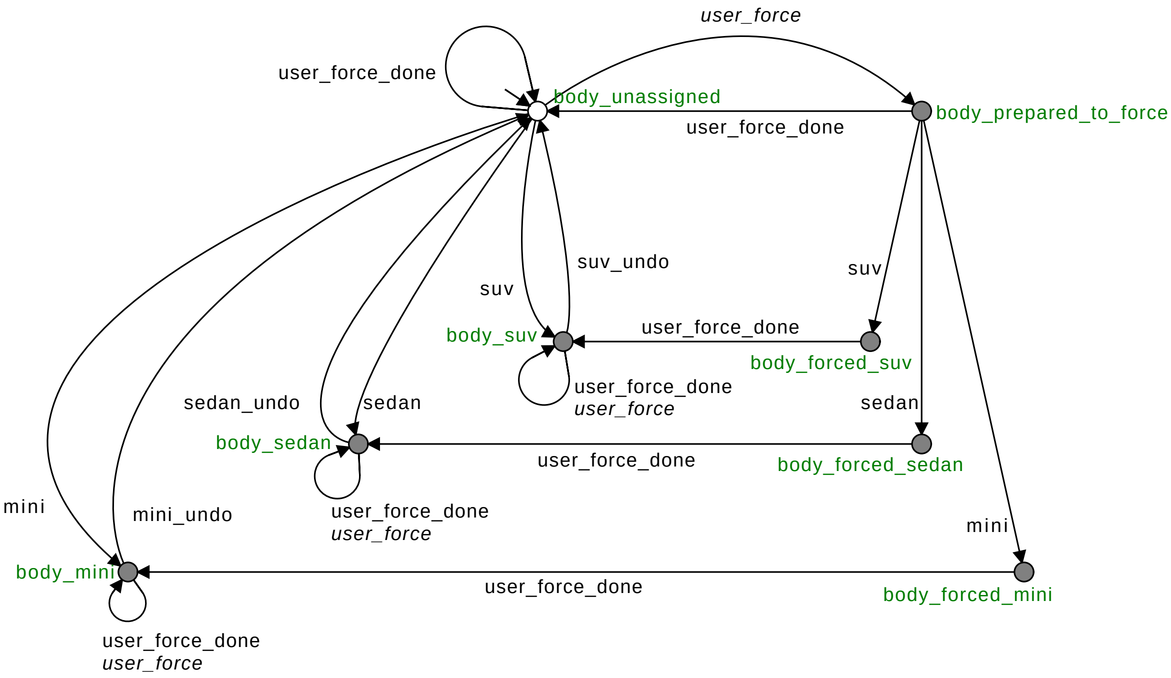


Figure 5.11: Automaton for variable *Body* that ensures that the enforced value is not offered for immediate undo (compare to Figure 5.5).

clause c_1 is already satisfied. The intuition behind the first guard is that, because x_3 is the last variable to be assigned, if by the time of choosing a value for x_3 clause c_2 is not satisfied yet, the only way to satisfy it is to choose x_3^F, thus to choose x_3^T clause c_2 should be satisfied by the other variables.

Similarly, the guards can be computed for the supervisor shown in Figure 5.2; these guards are shown in Table 5.4. Note that the guards represent only what and when the supervisor disables (or enables, if such representation is more compact). The supervisor does not have to keep the information about the complete state-space of the synchronous composition of the original automata.

Another promising approach to compact supervisor representation is to use compositional synthesis algorithms (Mohajerani et al. 2011). These algorithms apply abstraction techniques to the automata, as well as use multiple automata to represent the synthesized supervisor; the resulting supervisor can thus be called an *abstracted modular supervisor*. Preliminary experiments with compositional algorithms indicate that it might be possible to create a supervisor for product configuration instances that are larger than the ones that can be handled by other SCT methods. Comparison with BDDs and other knowledge compilation methods is a topic of future work.

5.3 Conclusions

Modeling interactive product configuration problem as an SCT problem allows compilation of the configuration constraints into a supervisor compactly represented by either symbolic guards or an abstracted modular supervisor. These representations of the supervisor can potentially be more compact than the other knowledge compilation methods used for product configuration, for example BDDs, while still providing

Table 5.3: Guards for the supervisor shown in Figure 5.4.

Event	Guard
x_3^T	$s_{c_2}^s$
x_3^F	$s_{c_1}^s$

Table 5.4: Guards for the supervisor shown in Figure 5.2.

Event	Guard
x_1^T	$s_{c_1}^s \wedge s_{x_3}^a \wedge s_{c_2}^u$
x_1^F	$s_{c_1}^u \wedge s_{x_3}^a \wedge s_{x_2}^a$
x_2^T	$s_{c_1}^u \wedge s_{x_3}^a \wedge s_{x_1}^a$
x_2^F	true
x_3^T	$s_{c_1}^s \wedge s_{x_1}^a \wedge s_{c_2}^u$
x_3^F	$s_{c_1}^u \wedge s_{x_2}^a \wedge s_{x_1}^a$

polynomial-time in the size of the compiled representation answers to important product configuration queries.

A limitation of the presented reconfiguration solution is that it does not help the user to find the shortest (or optimal with respect to some other criterion) way to repair a configuration. Future work may be to introduce optimization. Optimization can be implemented by showing the user for each event the minimum number of steps necessary to reach a marked state, or simply disable by the supervisor all undo-choices that do not lie on any of the shortest paths to a marked state. Work on optimization is ongoing in the SCT community (Miremadi 2012), and the model presented here will be usable for that purpose.

Chapter 6

Summary of Appended Papers

Paper 1.

Alexey Voronov, Knut Åkesson, Anna Tidstam, Johan Malmqvist and Martin Fabian. *Toward better support for authoring and maintaining product configuration constraints.* Submitted, 2012.

Paper 1 introduces a number of methods to help engineers maintain, verify and analyse product configuration constraints, including a method for automatic detection of errors in constraints, a method for authoring constraints for mutually-exclusive items, methods for improving internal structure of constraints for their better maintainability, and a method for efficiently computing the effects of adding or removing constraints. Computational performance of the proposed methods is tested on configuration data from automotive industry, and the results show that the methods can work very efficiently on such data.

Paper 2.

Alexey Voronov, Knut Åkesson, Anna Tidstam and Johan Malmqvist. *Verification of Item Usage Rules in Product Configuration.* Proceedings of 9th International Conference on Product Lifecycle Management PLM-12, Montreal, Canada, 2012.

Paper 2 introduces problems engineers face when working with Item Usage Rules (IURs), which specify which items are included in a bill of materials for a customer order. The paper considers ensuring that exactly one item from a predefined set is included in each product, as well as the problem of rewriting an IUR without changing the configurations that the IUR covers.

Paper 3.

Alexey Voronov, Knut Åkesson and Fredrik Ekstedt. *Enumerating partial configurations.* Proceedings of Configuration Workshop at 22nd International Joint Conference on Artificial Intelligence IJCAI-11, Barcelona, Spain, 2011.

Paper 3 focuses on the problem of efficient enumeration of valid partial configurations, which is used to provide a scoped view of a product, as well as for supporting engineers in authoring IURs. The paper provides motivation, pedagogical examples, detailed algorithms and empirical evaluation of different approaches to compute valid partial configurations.

Paper 4.

Koen Claessen, Niklas Een, Mary Sheeran, Niklas Sörensson, Alexey Voronov and Knut Åkesson. *SAT-Solving in Practice, with a Tutorial Example from Supervisory Control.* Journal of Discrete Event Dynamic Systems 19(4), pp. 495–524, 2009.

Paper 4 gives an overview of SAT-solving, and introduces a SAT-solver based approach to SCT problems, providing methods for SCT verification of controllability and deadlock-freeness, as well as SAT-based iterative synthesis procedure for controllable and deadlock-free supervisor.

Chapter 7

Conclusions and Future Work

This thesis introduces a number of methods for automatic verification of product configuration constraints and for computational support of manual inspection of constraints. These methods may ease elimination of errors and speed up the development process of a product platform, which in turn can increase the competitiveness of a company.

The conclusions are grouped around the research question (RQ).

RQ-1. *What kind of computer support can be implemented to help engineers maintain, verify and analyze product configuration constraints?*

For automatic verification of product configuration constraints, Paper 1 proposes to use reference configurations and Paper 2 proposes to use sets of mutually exclusive items to verify variant constraints and Item Usage Rules (IURs) after each modification of the constraints, similar to running unit tests in software engineering, which can speed up the detection of errors. Paper 1 proposes a method that can be used for both verification and authoring of IURs; this method is based on using valid partial configurations to verify or author IURs. How to compute valid partial configurations efficiently in investigated in Paper 3. These methods for automatic verification can reduce the number of errors and speed up the process of developing correct configuration constraints.

For supporting manual inspection, this thesis introduced a method to automatically verify that an IUR can be rewritten using a given subset of product variables (families). Rewriting an IUR can be useful as a maintenance operation, and it is important that it does not introduce errors. Rewriting an IUR is just one operation from a larger class of *refactoring* operations that improve the structure of constraints, but do not change their external behavior or meaning. The thesis also proposes a number of methods for discovering refactoring opportunities. When changing constraints, it might be difficult to foresee the effects of the change. This thesis introduces a method for efficiently computing the configurations that become allowed or forbidden when constraints are added or removed; this can be used when an engineer adds or removes constraints to automatically notify all other engineers that are influenced by the change, which can greatly improve the communication between engineers in a concurrent engineering setting.

RQ-2. *How to enumerate valid partial configurations efficiently?*

Valid partial configurations for large industrial product configuration data can be efficiently enumerated using SAT-solvers, as introduced in Paper 3. sd-DNNF might also be used for rather large product configuration instances. Problem instances that could be handled by sd-DNNF are larger than those that could be handled by MDDs, but not as large as those that could be handled by SAT. Performance of the proposed SCT-based method (Chapter 5) for enumerating valid partial configurations requires further investigations.

RQ-3. *How to compactly represent product configuration data for answering product configuration questions efficiently?*

SAT-solver with incremental solving can be very efficient in handling large product configuration problems from automotive industry. However, it is difficult to predict their running time, and there are no acceptable worst-case running time guarantees. Knowledge compilation methods can be used to create data structures that would provide acceptable guarantees on the worst-case running time. Among existing knowledge compilation methods for product configuration, sd-DNNF, AND/OR MDDs and Tree-of-BDDs look the most promising, although further investigations and benchmarking are necessary (see Chapter 4). We proposed a new knowledge compilation method for interactive product configuration based on Supervisory Control Theory (Chapter 5), but its applicability for large product configuration instances and benchmarking against other methods remains a topic of future work.

To summarize, this thesis introduces a number of formal methods for handling large-scale product configuration data. These methods can reduce the errors in creating and maintaining product configuration constraints, and speed up the development of product platforms. This thesis also opens up new questions for future work outlined below.

Future Work

One of the industrial partners in the research project adopted SAT-based methods for enumerating valid partial configurations described in Paper 3. It would be interesting to conduct a follow-up study to find out how helpful the methods are for real engineers, what improvements can be made to the methods, and whether more companies might benefit from deploying similar tools.

This thesis introduced a method for verifying that an IUR can be rewritten using a given set of families. This verification can be used within the optimization procedure. Such optimization procedure can automatically find the best set of families with respect to some criteria, for example, some engineers might want to have as many families as possible in an IUR to see all relevant details, while other engineers might prefer as few families as possible, to keep only essential information. Automatic rewriting can contribute to sustainability of the workplace by allowing engineers to chose their preferred representation method.

Verification and analysis queries in this thesis were formulated and programmed by researchers based on informal descriptions provided by engineers. To shorten the time from an idea to an implementation, and to improve the methods' adoption, an investigation of user-friendly methods to create queries about configuration constraints is needed, with the aim, for example, to create a language for constraint queries similar to database query languages, to be directly usable by engineers.

This thesis treated product configuration constraints as logic only. Taking into account the sources of constraints—for example, that some constraints are geometrical, and some constraints originate from marketing—might allow richer feedback to engineers, to facilitate understanding and improve decisions (for example, an engineer can relax marketing constraint much easier than a physical constraint). Further investigation is needed on how to connect such meta-data with constraints, and how to use it to create better explanations, and which other benefits it might bring and at which cost.

The methods presented in this thesis might be used to efficiently compute visualizations of configuration data similar to the ones presented in (Tidstam 2012; Tidstam, Bligård, et al. 2012; Pleuss et al. 2011; Ziyang 2010; Hadzic and O'Sullivan 2008). However, how to do that and which visualizations can be computed efficiently is not clear. Visualization methods might be very important when dealing with decisions that can not yet be automated (Baumeister and Freiberg 2010).

This thesis compares performance of some SAT, BDD, MDD and sd-DNNF tools, while other knowledge compilation methods are not benchmarked, including Tree-of-BDDs, AND/OR MDDs, as well as the SCT-based representation proposed in this thesis. Comparing as many existing tools as possible on a wide range of representative benchmarks could allow finding the best method for working with product configuration data.

This thesis attempts to find fixed-parameter-tractable properties in industrial product configuration instances. However, not all known tractable classes are investigated, and the number of industrial instances in the study was limited. As future work, one might extend the study with other tractable classes, for example, single lookahead unit resolution (SLUR) (Franco and Van Gelder 2003; Čepek et al. 2012) and matched formulas (Franco and Van Gelder 2003; Szeider 2003), as well as with more instances from other companies. Finding suitable fixed-parameter-tractable properties for industrial instances would allow improving and specializing both search and knowledge compilation procedures, which would result in faster answers.

Explanations of unsatisfiability in this thesis are based on extracting a Minimal Unsatisfiable Subformula (see Paper 1). However, this extraction does not take into account how complicated the resulting formula is for an engineer. Exploring human-friendly presentations of unsatisfiability and human understanding of formulas, for example, as in (Strannegård et al. 2009), could allow better explanations that can speed up comprehension and save engineers' time.

To make SAT-based algorithms more efficient for SCT problems as presented in Paper 4, one might start by looking at symmetry breaking (see, for example, (Sakallah

2009) for an introduction to symmetry and satisfiability), better encodings, and new verification approaches. Symmetry can be exemplified by multiple paths leading to the same state, while to verify a system it is often enough to analyze only one path. Symmetry can be avoided, for example, by adding extra constraints (Crawford et al. 1996), or by using solvers that take symmetry into account (Sabharwal 2009). Better encodings can also speed up the solving process. SAT solvers tend to work well with encodings that allow to achieve Generalized Arc Consistency via Unit Propagation only (Walsh 2000; Bacchus 2007). Such encodings exist for both automata transitions (for example, *grammar* constraint from (Bacchus 2007)) and for cardinality constraints that specify mutual exclusiveness of automata states (Bailleux and Boufkhad 2003; Sinz 2005; Marques-Silva and Lynce 2007; Bailleux 2010; Frisch and Giannaros 2010; Ben-Haim et al. 2012). Previously, Linear Temporal Logic (LTL) properties were the focus for SAT-based tools, and it is not possible to encode the non-blocking property using LTL. Recently, an approach to use SAT-solvers to check Computation Tree Logic (CTL) properties was proposed (Hassan et al. 2012), and CTL can be used to express non-blocking (Kumar and Jiang 2002). Using this approach for verifying CTL properties can allow verifying the non-blocking property and synthesizing non-blocking supervisors using SAT-solvers.

To summarize, there is a number of directions that can be pursued starting from this thesis to help engineers working with product configuration constraints.

Bibliography

Abate, Pietro, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli (Oct. 2012). "Dependency solving: A separate concern in component evolution management". In: *Journal of Systems and Software* 85.10, pp. 2228–2240. ISSN: 01641212. DOI: 10.1016/j.jss.2012.02.018 (cit. on p. 19).

Akers, Sheldon B (June 1978). "Binary Decision Diagrams". In: *IEEE Transactions on Computers* C-27.6, pp. 509–516. ISSN: 0018-9340. DOI: 10.1109/TC.1978.1675141 (cit. on p. 39).

Alves, Vander, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena (2006). "Refactoring product lines". In: *Proceedings of the 5th international conference on Generative programming and component engineering - GPCE '06*. New York, New York, USA: ACM Press, pp. 201–210. ISBN: 1595932372. DOI: 10.1145/1173706.1173737 (cit. on p. 4).

Amilhastre, Jérôme, Hélène Fargier, and Pierre Marquis (Feb. 2002). "Consistency restoration and explanations in dynamic CSPs – Application to configuration". In: *Artificial Intelligence* 135.1-2, pp. 199–234. ISSN: 00043702. DOI: 10.1016/S0004-3702(01)00162-X (cit. on pp. 4, 21, 40, 42, 56).

Amnell, Tobias and Pontus Jansson (2001). *Report from ASTEC-RT Auto project*. Tech. rep. Mecel AB, p. 11. URL: http://www.mecel.se/about/papers/ASTEC-RT-AUTO-report_final.pdf (cit. on p. 21).

Angele, J, D Fensel, D Landes, and Rudi Studer (1998). "Developing knowledge-based systems with MIKE". In: *Automated Software Engineering* 5.4, pp. 389–418. DOI: 10.1023/A:1008653328901 (cit. on p. 7).

Apt, Krzysztof (2003). *Principles of Constraint Programming*. Cambridge University Press, p. 420. ISBN: 9780521825832 (cit. on pp. 5, 24).

Asikainen, Timo, Tomi Männistö, and Timo Soininen (Jan. 2007). "Kumbang: A domain ontology for modelling variability in software product families". In: *Advanced Engineering Informatics* 21.1, pp. 23–40. ISSN: 14740346. DOI: 10.1016/j.aei.2006.11.007 (cit. on p. 4).

Aspvall, Bengt and MF Plass (1979). "A linear-time algorithm for testing the truth of certain quantified boolean formulas". In: *Information Processing Letters* 8.3, pp. 121–123 (cit. on pp. 22, 50).

Astesana, Jean-Marc, Yves Bossu, Laurent Cosserat, and Hélène Fargier (2010). "Constraint-based Modeling and Exploitation of a Vehicle Range at Renault's: Requirement analysis and complexity study". In: *ECAI 2010 Workshop on Configuration*, pp. 33–39 (cit. on pp. 4, 13, 21).

Astesana, Jean-Marc, Laurent Cosserat, and Hélène Fargier (Oct. 2010). "Constraint-based Vehicle Configuration: A Case Study". In: *22nd IEEE International Conference on Tools with Artificial Intelligence ICTAI 2010*. IEEE, pp. 68–75. ISBN: 978-1-4244-8817-9. DOI: 10.1109/ICTAI.2010.19 (cit. on pp. 4, 11, 13, 21).

Bacchus, Fahiem (2007). "GAC Via Unit Propagation". In: *13th International Conference on Principles and Practice of Constraint Programming, CP 2007*. Ed. by Christian Bessière. Vol. 4741. LNCS. Providence, RI, USA: Springer, pp. 133–147. DOI: 10.1007/978-3-540-74970-7_12 (cit. on p. 76).

Bailleux, Olivier (2010). *On the CNF encoding of cardinality constraints and beyond.* Tech. rep., pp. 1–8. arXiv:arXiv:1012.3853v1 (cit. on p. 76).

Bailleux, Olivier and Yacine Boufkhad (Oct. 2003). "Efficient CNF Encoding of Boolean Cardinality Constraints". In: *9th International Conference on Principles and Practice of Constraint Programming - CP 2003*. Ed. by F. Rossi. Vol. 2833. LNCS. Kinsale, Ireland: Springer, pp. 108–122. DOI: 10.1007/978-3-540-45193-8_8 (cit. on p. 76).

Barker, Virginia E., Dennis E. O'Connor, Judith Bachant, and Elliot Soloway (Mar. 1989). "Expert systems for configuration at Digital: XCON and beyond". In: *Communications of the ACM* 32.3, pp. 298–318. ISSN: 00010782. DOI: 10.1145/62065.62067 (cit. on p. 4).

Batory, Don S. (2005). "Feature Models, Grammars, and Propositional Formulas". In: *9th International Conference on Software Product Lines*. Ed. by Henk Obbink and Klaus Pohl. Vol. 3714. LNCS. Rennes, France: Springer, pp. 7 –20. DOI: 10.1007/11554844_3 (cit. on pp. 4, 13).

Baumeister, Joachim and Martina Freiberg (Oct. 2010). "Knowledge visualization for evaluation tasks". In: *Knowledge and Information Systems* 29.2, pp. 349–378. ISSN: 0219-1377. DOI: 10.1007/s10115-010-0350-8 (cit. on pp. 4, 75).

Baumeister, Joachim, Frank Puppe, and Dietmar Seipel (2004). "Refactoring methods for knowledge bases". In: *Engineering Knowledge in the Age of the Semantic Web*. Springer, pp. 157–171. DOI: 10.1007/978-3-540-30202-5_11 (cit. on p. 18).

Benavides, David, Sergio Segura, and Antonio Ruiz-Cortés (Sept. 2010). "Automated analysis of feature models 20 years later: A literature review". In: *Information Systems* 35.6, pp. 615–636. ISSN: 03064379. DOI: 10.1016/j.is.2010.01.001 (cit. on pp. 4, 13).

Ben-Haim, Yael, Alexander Ivrii, Oded Margalit, and Arie Matsliah (2012). "Perfect Hashing and CNF Encodings of Cardinality Constraints". In: *15th International Conference on Theory and Applications of Satisfiability Testing – SAT 2012*. Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. LNCS. Trento, Italy: Springer, pp. 397–409. DOI: 10.1007/978-3-642-31612-8_30 (cit. on pp. 27, 76).

Bennaceur, Hachemi (Apr. 2004). "A Comparison between SAT and CSP Techniques". In: *Constraints* 9.2, pp. 123–138. ISSN: 1383-7133. DOI: 10.1023/B:CONS.0000024048.03454.c0 (cit. on p. 43).

Biere, Armin, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu (1999). "Symbolic Model Checking without BDDs". In: *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'99*.

97. Amsterdam, The Netherlands: Springer, pp. 193–207. DOI: `10.1007/3-540-49059-0_14` (cit. on p. 29).

Biere, Armin, Marijn Heule, Hans van Maaren, and Toby Walsh, eds. (Feb. 2009). *Handbook of Satisfiability.* Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, p. 980. ISBN: 978-1-58603-929-5 (cit. on p. 5).

Biere, Armin and W. Kunz (2002). "SAT and ATPG: Boolean engines for formal hardware verification". In: *IEEE/ACM International Conference on Computer Aided Design. ICCAD-2002.* IEEE, pp. 782–785. ISBN: 0-7803-7607-2. DOI: `10.1109/ICCAD.2002.1167620` (cit. on p. 27).

Boehm, B.W. (Jan. 1984). "Verifying and Validating Software Requirements and Design Specifications". In: *IEEE Software* 1.1, pp. 75–88. ISSN: 0740-7459. DOI: `10.1109/MS.1984.233702` (cit. on p. 12).

Bollig, Beate and Ingo Wegener (1996). "Improving the variable ordering of OBDDs is NP-complete". In: *IEEE Transactions on Computers* 45.9, pp. 993–1002. ISSN: 00189340. DOI: `10.1109/12.537122` (cit. on p. 40).

Bordeaux, Lucas, Youssef Hamadi, and Lintao Zhang (2006). "Propositional Satisfiability and Constraint Programming: A comparative survey". In: *ACM Computing Surveys* 38.4, pp. 1–62. ISSN: 0360-0300. DOI: `10.1145/1177352.1177354` (cit. on p. 43).

Bradley, Aaron R (2011). "SAT-Based Model Checking without Unrolling". In: *12th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI 2011.* Ed. by Ranjit Jhala and David Schmidt. Austin, Texas: Springer, pp. 70–87. DOI: `10.1007/978-3-642-18275-4_7` (cit. on p. 48).

Bradley, Aaron R and Zohar Manna (Nov. 2007). "Checking Safety by Inductive Generalization of Counterexamples to Induction". In: *Formal Methods in Computer Aided Design (FMCAD'07).* Austin, Texas: IEEE, pp. 173–180. ISBN: 0-7695-3023-0. DOI: `10.1109/FAMCAD.2007.15` (cit. on p. 48).

Bruynooghe, Maurice (Feb. 1981). "Solving combinatorial search problems by intelligent backtracking". In: *Information Processing Letters* 12.1, pp. 36–39. ISSN: 00200190. DOI: `10.1016/0020-0190(81)90074-0` (cit. on p. 25).

Bryant, Randal E. (Aug. 1986). "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35.8, pp. 677–691. ISSN: 0018-9340. DOI: `10.1109/TC.1986.1676819` (cit. on pp. 5, 39, 40).

Bühne, Stan, Kim Lauenroth, and Klaus Pohl (2004). "Why is it not Sufficient to Model Requirements Variability with Feature Models?" In: *Automotive Requirements Engineering (AURE04)*, pp. 5–12 (cit. on p. 12).

Burch, Jerry R., Edmund M. Clarke, Ken L McMillan, David L. Dill, and L.J. Hwang (1990). "Symbolic model checking: 10E20 states and beyond". In: *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on.* IEEE, pp. 428–439. DOI: `10.1109/LICS.1990.113767` (cit. on p. 5).

Cadoli, Marco and Francesco M. Donini (1997). "A Survey on Knowledge Compilation". In: *AI Communications* 10.3-4, pp. 137–150. ISSN: 1875-8452 (cit. on p. 39).

Cassandras, Christos G. and Stephane Lafortune (Sept. 2008). *Introduction to Discrete Event Systems.* 2nd. Boston, MA: Springer US, p. 776. ISBN: 978-0-387-33332-8. DOI: `10.1007/978-0-387-68612-7` (cit. on pp. 29–32).

Čepek, Ondřej, Petr Kučera, and Václav Vlček (2012). "Properties of SLUR formulae". In: *38th Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM-2012*. Ed. by Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán. Vol. 7147. Špindleruv Mlýn, Czech Republic: Springer, pp. 177–189. DOI: 10.1007/978-3-642-27660-6_15 (cit. on p. 75).

Claessen, Koen, Niklas Een, Mary Sheeran, and Niklas Sörensson (2008). "SAT-solving in practice". In: *9th International Workshop on Discrete Event Systems WODES-2008*. Göteborg, Sweden: IEEE, pp. 61–67. ISBN: 978-1-4244-2592-1. DOI: 10.1109/WODES.2008.4605923 (cit. on p. 38).

Clarke, Edmund M., Armin Biere, Richard Raimi, and Yunshan Zhu (July 2001). "Bounded Model Checking Using Satisfiability Solving". In: *Form. Methods Syst. Des.* 19.1, pp. 7–34. ISSN: 0925-9856. DOI: 10.1023/A:1011276507260 (cit. on p. 48).

Clarke, Edmund M., Orna Grumberg, and Doron A. Peled (2000). *Model Checking*. MIT Press, p. 330. ISBN: 978-0-262-03270-4 (cit. on p. 21).

Cook, Stephen A. (1971). "The complexity of theorem-proving procedures". In: *Proceedings of the third annual ACM symposium on*. ACM Press, pp. 151–158. DOI: 10.1145/800157.805047 (cit. on pp. 5, 22, 30).

Crawford, James M., Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy (1996). "Symmetry-Breaking Predicates for Search Problems". In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*. Ed. by Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro. Cambridge, Massachusetts, USA: Morgan Kaufmann, pp. 148–159 (cit. on p. 76).

Crow, Judith and John Rushby (1994). *Model-based reconfiguration: Diagnosis and recovery*. Tech. rep. NASA, p. 59. URL: http://ntrs.nasa.gov/search.jsp?R=19940028270 (cit. on p. 18).

Darwiche, Adnan (1998). "Model-Based Diagnosis using Structured System Descriptions". In: *Journal of Artificial Intelligence Research* 8, pp. 165–222 (cit. on pp. 40, 41).

Darwiche, Adnan (1999). "Compiling knowledge into decomposable negation normal form". In: *IJCAI 1999*. Vol. 16. Citeseer, pp. 284–289 (cit. on p. 42).

Darwiche, Adnan (July 2001a). "Decomposable negation normal form". In: *Journal of the ACM* 48.4, pp. 608–647. ISSN: 00045411. DOI: 10.1145/502090.502091 (cit. on pp. 40, 41).

Darwiche, Adnan (Jan. 2001b). "On the Tractable Counting of Theory Models and its Application to Truth Maintenance and Belief Revision". In: *Journal of Applied Non-Classical Logics* 11.1-2, pp. 11–34. ISSN: 1166-3081. DOI: 10.3166/jancl.11.11-34. arXiv:0003044 [cs] (cit. on pp. 42, 49).

Darwiche, Adnan (2004). "New Advances in Compiling CNF to Decomposable Negation Normal Form". In: *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004*. Ed. by Ramon López de Mántaras and Lorenza Saitta. Vol. 110. Frontiers in Artificial Intelligence and Applications. Valencia, Spain: IOS Press, pp. 328–332 (cit. on pp. 42, 46).

Darwiche, Adnan and Pierre Marquis (2002). “A knowledge compilation map”. In: *Journal of Artificial Intelligence Research* 17.1, pp. 229–264. DOI: 10.1613/jair.989 (cit. on pp. 39–41).

Davis, Martin, George Logemann, and Donald Loveland (July 1962). “A machine program for theorem-proving”. In: *Communications of the ACM* 5.7, pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557 (cit. on p. 26).

Davis, Martin and Hilary Putnam (July 1960). “A Computing Procedure for Quantification Theory”. In: *Journal of the ACM* 7.3, pp. 201–215. ISSN: 00045411. DOI: 10.1145/321033.321034 (cit. on pp. 5, 26).

Davis, Randall (Dec. 1984). “Diagnostic reasoning based on structure and behavior”. In: *Artificial Intelligence* 24.1-3, pp. 347–410. ISSN: 00043702. DOI: 10.1016/0004-3702(84)90042-0 (cit. on p. 25).

Davis, Stanley M. (1987). *Future perfect*. Reading, MA: Addison-Wesley, p. 243. ISBN: 9780201327953 (cit. on p. 3).

Dechter, Rina (1986). “Learning while searching in constraint-satisfaction-problems”. In: *AAAI 1986*, pp. 178–183 (cit. on p. 25).

Dechter, Rina (Jan. 1990). “Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition”. In: *Artificial Intelligence* 41.3, pp. 273–312. ISSN: 00043702. DOI: 10.1016/0004-3702(90)90046-3 (cit. on p. 25).

Dechter, Rina (2003). *Constraint Processing*. Morgan Kaufmann (Elsevier), p. 480. ISBN: 978-1-55860-890-0 (cit. on p. 5).

Dechter, Rina and Judea Pearl (Dec. 1987). “Network-based heuristics for constraint-satisfaction problems”. In: *Artificial Intelligence* 34.1, pp. 1–38. ISSN: 00043702. DOI: 10.1016/0004-3702(87)90002-6 (cit. on p. 67).

Dechter, Rina and Judea Pearl (Apr. 1989). “Tree clustering for constraint networks”. In: *Artificial Intelligence* 38.3, pp. 353–366. ISSN: 00043702. DOI: 10.1016/0004-3702(89)90037-4 (cit. on pp. 40, 42, 43).

Desharnais, Jean-Marc, Alain Abran, and Julien Vilz (2011). “Verification and validation of a knowledge-based system”. In: *Common Software Measurement International Consortium (COSMIC) Function Point: Theory and Advanced Practices*. Ed. by Reiner Dumke and Alain Abran. Taylor and Francis. Chap. 5.2. ISBN: 978-1-43-984486-1 (cit. on pp. 4, 21).

Dilkina, Bistra, Carla P. Gomes, and Ashish Sabharwal (2007). “Tradeoffs in the Complexity of Backdoor Detection”. In: *13th International Conference on Principles and Practice of Constraint Programming, CP 2007*. Ed. by Christian Bessière. Vol. 4741. LNCS. Providence, RI, USA: Springer Berlin Heidelberg, pp. 256–270. DOI: 10.1007/978-3-540-74970-7_20 (cit. on pp. 51, 52).

Dongen, Stijn van (2000). *A Cluster Algorithm for Graphs*. Tech. rep. Amsterdam: Centrum voor Wiskunde en Informatica. URL: http://oai.cwi.nl/oai/asset/4463/04463D.pdf (cit. on p. 45).

Dowling, William F and Jean H Gallier (1984). “Linear-time algorithms for testing the satisfiability of propositional Horn formulae”. In: *The Journal of Logic Programming*, pp. 267–284 (cit. on pp. 22, 50).

Downey, Rodney G. and Michael R. Fellows (1999). *Parameterized Complexity*. Springer-Verlag. ISBN: 978-0-387-94883-6 (cit. on p. 50).

Duffy, David A. (1991). *Principles of Automated Theorem Proving.* Wiley (cit. on p. 21).

Een, Niklas and Niklas Sörensson (Jan. 2003). "Temporal Induction by Incremental SAT Solving". In: *Electronic Notes in Theoretical Computer Science: Proceedings of First International Workshop on Bounded Model Checking* 89.4, pp. 543–560. ISSN: 15710661. DOI: 10.1016/S1571-0661(05)82542-3 (cit. on pp. 39, 48).

Een, Niklas and Niklas Sörensson (2004). "An Extensible SAT-solver". In: *Theory and Applications of Satisfiability Testing* 2919, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37 (cit. on p. 45).

Een, Niklas and Niklas Sörensson (2006). "Translating pseudo-boolean constraints into SAT". In: *Journal on Satisfiability, Boolean Modeling and Computation* 2, pp. 1–26 (cit. on p. 45).

Emerson, E. Allen and Edmund M. Clarke (1980). "Characterizing correctness properties of parallel programs using fixpoints". In: *Seventh Colloquium Noordwijkerhout on Automata, Languages and Programming.* Ed. by Jaco de Bakker and Jan van Leeuwen. Vol. 85. Lecture Notes in Computer Science. Springer, pp. 169–181. DOI: 10.1007/3-540-10003-2_69 (cit. on p. 21).

Eppstein, David (2007). *A graph and its tree decomposition.* URL: http://commons.wikimedia.org/wiki/File:Tree_decomposition.svg (visited on 11/19/2012) (cit. on p. 51).

Falkner, Andreas A and Alois Haselböck (2010). "Challenges of Knowledge Evolution in Practice". In: *Configuration Workshop at ECAI 2010.* Lisbon, Portugal, p. 71 (cit. on p. 18).

Fargier, Hélène and Pierre Marquis (2009). "Knowledge compilation properties of Trees-of-BDDs, revisited". In: *Proc. of IJCAI'09*, pp. 772–777 (cit. on pp. 40, 43).

Fargier, Hélène and Marie-Catherine Vilarem (Oct. 2004). "Compiling CSPs into Tree-Driven Automata for Interactive Solving". In: *Constraints* 9.4, pp. 263–287. ISSN: 1383-7133. DOI: 10.1023/B:CONS.0000049204.75635.7e (cit. on pp. 40, 42, 43).

Felfernig, Alexander, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner (Feb. 2004). "Consistency-based diagnosis of configuration knowledge bases". In: *Artificial Intelligence* 152.2, pp. 213–234. ISSN: 00043702. DOI: 10.1016/S0004-3702(03)00117-6 (cit. on pp. 13, 18).

Felfernig, Alexander, Gerhard Friedrich, Monika Schubert, Monika Mandl, Markus Mairitsch, and Erich Teppan (2009). "Plausible repairs for inconsistent requirements". In: *Proceedings of the 21st international jont conference on Artifical intelligence IJCAI-2009.* Morgan Kaufmann Publishers Inc., pp. 791–796 (cit. on p. 18).

Flum, Jörg and Martin Grohe (2006). *Parameterized Complexity Theory.* Texts in Theoretical Computer Science. An EATCS Series. Berlin, Heidelberg: Springer Berlin Heidelberg, p. 493. ISBN: 978-3-540-29952-3. DOI: 10.1007/3-540-29953-X (cit. on p. 50).

Fogliatto, Flávio S, Giovani J.C. da Silveira, and Denis Borenstein (Mar. 2012). "The mass customization decade: An updated review of the literature". In: *International*

Journal of Production Economics 138.1, pp. 14–25. ISSN: 09255273. DOI: 10.1016/j.ijpe.2012.03.002 (cit. on p. 3).

Ford, Henry (1922). *My Life and Work*. New York, NY, USA: Garden City Publishing Company, Inc (cit. on p. 3).

Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts (July 1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional, p. 464. ISBN: 978-0201485677 (cit. on p. 17).

Franco, John and Allen Van Gelder (Feb. 2003). “A perspective on certain polynomial-time solvable classes of satisfiability”. In: *Discrete Applied Mathematics* 125.2-3, pp. 177–214. ISSN: 0166218X. DOI: 10.1016/S0166-218X(01)00358-4 (cit. on p. 75).

Freuder, Eugene C. (Nov. 1978). “Synthesizing constraint expressions”. In: *Communications of the ACM* 21.11, pp. 958–966. ISSN: 00010782. DOI: 10.1145/359642.359654 (cit. on p. 25).

Freuder, Eugene C. and Alan K Mackworth (2006). “Constraint Satisfaction: An Emerging Paradigm”. In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Elsevier. Chap. 2, pp. 13–27. DOI: 10.1016/S1574-6526(06)80006-4 (cit. on p. 22).

Friedrich, Gerhard, Anna Ryabokon, Andreas A Falkner, Alois Haselböck, Gottfried Schenner, and Herwig Schreiner (2011). “(Re)configuration using Answer Set Programming”. In: *Workshop on Configuration In conjunction with the 22nd International Joint Conference on Artificial Intelligence - IJCAI 2011*. Ed. by Kostyantyn Shchekotykhin, Dietmar Jannach, and Markus Zanker. Vol. 755. Barcelona, Spain: CEUR-WS (cit. on p. 18).

Frisch, Alan M. and Paul A. Giannaros (2010). “SAT Encodings of the At-Most-k Constraint”. In: *The 9th International Workshop on Constraint Modelling and Reformulation (ModRef 2010) at the 16th International Conference on the Principles and Practice of Constraint Programming (CP 2010)* (cit. on pp. 27, 76).

Fuxin, Freddy (2005). “Evolution and communication of geometry based product information within an extended enterprise (PhD thesis)”. PhD thesis. Luleåtekniska universitet. URL: http://epubl.ltu.se/1402-1544/2005/04/ (cit. on pp. 4, 12).

Gardner, Martin (1958). *Logic machines and diagrams*. McGraw-Hill (cit. on pp. 5, 23).

Gelle, Esther and Rainer Weigel (Dec. 1996). “Interactive configuration using constraint satisfaction techniques”. In: *AAAI Fall Symposia ’96*. Cambridge, Massachusetts, USA, pp. 37–44 (cit. on p. 19).

Gent, Ian P and Peter Nightingale (2004). “A New Encoding of AllDifferent into SAT”. In: *Modelling and Reformulating Constraint Satisfaction Problems: Towards Systematisation and Automation*. Ed. by Alan M. Frisch and Ian Miguel, pp. 95–110 (cit. on p. 28).

Gohari, P. and W. Murray Wonham (2000). “On the complexity of supervisory control design in the RW framework”. In: *Systems, Man, and Cybernetics, Part B, IEEE Transactions on* 30.5, pp. 643–652 (cit. on p. 30).

Golomb, Solomon W. and Leonard D. Baumert (Oct. 1965). “Backtrack Programming”. In: *Journal of the ACM* 12.4, pp. 516–524. ISSN: 00045411. DOI: 10.1145/321296.321300 (cit. on pp. 22, 25).

Gottlob, Georg, Francesco Scarcello, and Martha Sideri (June 2002). “Fixed-parameter complexity in AI and nonmonotonic reasoning”. In: *Artificial Intelligence* 138.1-2, pp. 55–86. ISSN: 00043702. DOI: 10.1016/S0004-3702(02)00182-0 (cit. on p. 52).

Gupta, Uma G. (Dec. 1993). “Validation and verification of knowledge-based systems: A survey”. In: *Applied Intelligence* 3.4, pp. 343–363. ISSN: 0924-699X. DOI: 10.1007/BF00872136 (cit. on pp. 4, 21).

Haag, Albert (July 1998). “Sales configuration in business processes”. In: *IEEE Intelligent Systems* 13.4, pp. 78–85. ISSN: 1094-7167. DOI: 10.1109/5254.708436 (cit. on p. 9).

Hadzic, Tarik and Henrik Reif Andersen (2005). “Interactive Reconfiguration in Power Supply Restoration”. In: *11th International Conference on Principles and Practice of Constraint Programming CP-2005*. Ed. by Peter van Beek. Vol. 3709. LNCS. Sitges, Spain: Springer, pp. 767–771. DOI: 10.1007/11564751_61 (cit. on p. 19).

Hadzic, Tarik and Esben Rune Hansen (2008). “On Automata, MDDs and BDDs in Constraint Satisfaction”. In: *Workshop on Inference methods based on Graphical Structures of Knowledge at ECAI’08*. Ed. by Rina Dechter, Hélène Fargier, Jürg Kohlas, Jérôme Mengin, Gérard Verfaillie, and Nic Wilson. Patras, Greece (cit. on pp. 40, 42).

Hadzic, Tarik, Esben Rune Hansen, and Barry O’Sullivan (Nov. 2008). “Layer Compression in Decision Diagrams”. In: *20th IEEE International Conference on Tools with Artificial Intelligence*. c. Dayton, OH: IEEE, pp. 19–26. ISBN: 978-0-7695-3440-4. DOI: 10.1109/ICTAI.2008.92 (cit. on p. 55).

Hadzic, Tarik, John N. Hooker, Barry O’Sullivan, and Peter Tiedemann (2008). “Approximate Compilation of Constraints into Multivalued Decision Diagrams”. In: *14th International Conference on Principles and Practice of Constraint Programming, CP-2008*. Ed. by Peter J. Stuckey. Vol. 5202. LNCS. Sydney, Australia: Springer. DOI: 10.1007/978-3-540-85958-1_30 (cit. on p. 42).

Hadzic, Tarik and Barry O’Sullivan (July 2008). “Beyond Valid Domains in Interactive Configuration”. In: *Configuration Workshop at ECAI 2008* (cit. on p. 75).

Hadzic, Tarik, Sathiamoorthy Subbarayan, Rune Møller Jensen, Henrik Reif Andersen, Jesper Møller, and H. Hulgaard (2004). “Fast backtrack-free product configuration using a precompiled solution space representation”. In: *PETO conference* (cit. on pp. 19, 40).

Hamadi, Youssef and Lucas Bordeaux (May 2007). “On the First SAT/CP Integration Workshop”. In: *Trends in Constraint Programming*. Ed. by Frédéric Benhamou, Narendra Jussien, and Barry O’Sullivan. ISTE. Chap. 5, pp. 105–123. ISBN: 978-1-905209-97-2. DOI: 10.1002/9780470612309.ch5 (cit. on p. 43).

Hamscher, Walter (July 1992). “Model-based reasoning in financial domains”. In: *The Knowledge Engineering Review* 7.04, p. 323. ISSN: 0269-8889. DOI: 10.1017/S0269888900006457 (cit. on p. 4).

Hansen, Esben Rune and Peter Tiedemann (2007). “Compressing Configuration Data for Memory Limited Devices”. In: *AAAI 2007*, pp. 210–216 (cit. on p. 55).

Hassan, Zyad, Aaron R Bradley, and Fabio Somenzi (2012). “Incremental, Inductive CTL Model Checking”. In: *24th International Conference on Computer Aided Verification CAV-2012*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. LNCS. Berkeley, CA: Springer, pp. 532–547. DOI: `10.1007/978-3-642-31424-7_38` (cit. on p. 76).

Hayes-Roth, Frederick (Sept. 1985). “Rule-based systems”. In: *Communications of the ACM* 28.9, pp. 921–932. ISSN: 00010782. DOI: `10.1145/4284.4286` (cit. on p. 4).

Hertli, Timon, Robin A Moser, and Dominik Scheder (2011). “Improving PPSZ for 3-SAT using Critical Variables”. In: *28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*. Ed. by Thomas Schwentick and Christoph Dürr. Vol. 9. LIPICS. Dortmund, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 237–248. ISBN: 978-3-939897-25-5. DOI: `10.4230/LIPIcs.STACS.2011.237` (cit. on pp. 5, 22, 50).

Hoffmann, Gerard and Howard Wong-Toi (1992). “Symbolic synthesis of supervisory controllers”. In: *American Control Conference, 1992*. Chicago, IL, USA: IEEE, pp. 2789–2793 (cit. on p. 56).

Hopcroft, John E. and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley. ISBN: 81-7808-347-7 (cit. on p. 56).

Hutter, Frank, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle (2009). “ParamILS: An Automatic Algorithm Configuration Framework”. In: *Journal of Artificial Intelligence Research* 36.1, pp. 267–306. ISSN: 10769757. DOI: `10.1613/jair.2861` (cit. on p. 46).

Hvam, Lars, Niels Henrik Mortensen, and Jesper Riis (2008). *Product Customization*. Springer, p. 283. ISBN: 978-3-540-71448-4 (cit. on p. 3).

Iwama, Kazuo and Shuichi Miyazaki (1994). “SAT-variable complexity of hard combinatorial problems”. In: *IFIP World Computer Congress*, pp. 253–258 (cit. on p. 27).

Janota, Mikolas (2008). “Do SAT solvers make good configurators?” In: *12th International Software Product Line Conference. First Workshop on Analyses of Software Product Lines*, pp. 1–5 (cit. on p. 39).

Janota, Mikolas (2010). “SAT Solving in Interactive Configuration (PhD thesis)”. PhD thesis. University College Dublin (cit. on pp. 19, 39).

Jensen, Rune Møller (2004a). *CLab 1.0 User Manual*. Tech. rep. CMU. URL: `http://www.cs.cmu.edu/$\sim$runej/data/systems/clab10/man.pdf` (cit. on p. 45).

Jensen, Rune Møller (2004b). “CLab: a C++ Library for Fast Backtrack-Free Interactive Product Configuration”. In: *10th International Conference on Principles and Practice of Constraint Programming, CP-2004*. Ed. by Mark Wallace. Vol. 3258. Lecture Notes in Computer Science. Toronto, Ontario, Canada: Springer, p. 816. DOI: `10.1007/978-3-540-30201-8_94` (cit. on p. 45).

Johansen, Karsten Friis and Henrik Rosenmeier (1998). *A History of Ancient Philosophy: From the Beginnings to Augustine*. London: Routledge. ISBN: 0-415-12738-6 (cit. on pp. 5, 23).

Junker, Ulrich (Aug. 2006). “Configuration”. In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Foundations of Artificial

Intelligence. Elsevier Science Inc. Chap. 24, pp. 837–874. DOI: 10.1016/S1574-6526(06)80028-3 (cit. on p. 4).

Junker, Ulrich and Daniel Mailharro (Aug. 2003). “Preference programming: Advanced problem solving for configuration”. In: *AI EDAM: Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 17.01, pp. 13–29. ISSN: 0890-0604. DOI: 10.1017/S089006040317103X (cit. on p. 4).

Kam, T., T. Villa, R. Brayton, and A Sangiovanni-Vincentelli (1998). “Multi-valued decision diagrams: theory and applications”. In: *Multiple-Valued Logic* 4, pp. 9–62 (cit. on p. 40).

Kang, Kyo C, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Tech. rep. November. Pittsburgh, PA, USA: Carnegie Mellon University, Software Engineering Institute, p. 163. URL: http://www.sei.cmu.edu/reports/90tr021.pdf (cit. on pp. 4, 12).

Kimura, Shinji and Edmund M. Clarke (1990). “A parallel algorithm for constructing binary decision diagrams”. In: *Proceedings., 1990 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 220–223. DOI: 10.1109/ICCD.1990.130209 (cit. on p. 56).

Kleer, Johan de (1989). “A comparison of ATMS and CSP techniques”. In: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89.* Vol. 1, pp. 290–296 (cit. on p. 27).

Kolodner, Janet L. (1992). “An introduction to case-based reasoning”. In: *Artificial Intelligence Review* 6.1, pp. 3–34. ISSN: 0269-2821. DOI: 10.1007/BF00155578 (cit. on p. 3).

Kottler, Stephan, Michael Kaufmann, and Carsten Sinz (May 2008). “Computation of renameable horn backdoors”. In: *11th International Conference on Theory and Applications of Satisfiability - SAT 2008.* Ed. by Hans Kleine Büning and Xishun Zhao. Vol. 4996. LNCS. Guangzhou, China: Springer, pp. 154–160. ISBN: 978-1-4244-5821-9. DOI: 10.1007/978-3-540-79719-7_15 (cit. on p. 52).

Kreuz, Ingo and Dieter Roller (1999). “Knowledge Growing Old in Reconfiguration Context”. In: *Configuration Workshop at AAAI 1999*, pp. 54–58 (cit. on p. 18).

Kübler, Andreas, Christoph Zengler, and Wolfgang Küchlin (July 2010). “Model Counting in Product Configuration”. In: *Electronic Proceedings in Theoretical Computer Science* 29.LoCoCo, pp. 44–53. ISSN: 2075-2180. DOI: 10.4204/EPTCS.29.5. arXiv:1007.0831 (cit. on pp. 11, 42).

Küchlin, Wolfgang and Carsten Sinz (Feb. 2000). “Proving Consistency Assertions for Automotive Product Data Management”. In: *Journal of Automated Reasoning* 24.1, pp. 145–163 (cit. on pp. 13, 21, 39, 44).

Kumar, Ratnesh and Shengbing Jiang (2002). “Supervisory control of discrete event systems with CTL* temporal logic specifications”. Orlando, FL. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=980826 (cit. on p. 76).

Le Berre, Daniel and Anne Parrain (2010). “The Sat4j library, release 2.2 system description”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7, pp. 59–64 (cit. on p. 45).

Lee, C.Y. (1959). “Representation of switching circuits by binary-decision programs”. In: *Bell System Technical Journal* 38.4, pp. 985–999 (cit. on p. 39).

Lewis, Harry R. (Jan. 1978). “Renaming a Set of Clauses as a Horn Set”. In: *Journal of the ACM* 25.1, pp. 134–135. ISSN: 00045411. DOI: 10.1145/322047.322059 (cit. on p. 50).

Lindahl, Magnus, Paul Pettersson, and Wang Yi (1998). “Formal Design and Analysis of a Gear Controller”. In: *4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS’98, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS’98.* Ed. by Bernhard Steffen. Vol. 1384. LNCS. Lisbon, Portugal: Springer, pp. 281–297. DOI: 10.1007/BFb0054178 (cit. on p. 21).

Lindahl, Magnus, Paul Pettersson, and Wang Yi (2001). “Formal design and analysis of a gear controller”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 3.3, pp. 353–368. DOI: 10.1007/s100090100048 (cit. on p. 21).

Lind-Nielsen, Jørn (2002). *BuDDy: A BDD package.* Miscellaneous. URL: http://buddy.sourceforge.net (cit. on p. 45).

Lindroth, Peter (2011). “Product Configuration from a Mathematical Optimization Perspective”. PhD thesis. Chalmers University of Technology and University of Gothenburg. ISBN: 9789173855341. URL: http://publications.lib.chalmers.se/publication/140022 (cit. on p. 4).

Li, Zijie and Peter van Beek (2011). “Finding Small Backdoors in SAT Instances”. In: *Proceedings of the 24th Canadian conference on Advances in artificial intelligence - Canadian AI’11.* Ed. by Cory Butz and Pawan Lingras. Vol. 6657/2011. Lecture Notes in Computer Science. St. John’s, Canada: Springer, pp. 269–280. DOI: 10.1007/978-3-642-21043-3_33 (cit. on p. 51).

Maaren, Hans van (May 2000). “A Short Note on Some Tractable Cases of the Satisfiability Problem”. In: *Information and Computation* 158.2, pp. 125–130. DOI: 10.1006/inco.2000.2867 (cit. on p. 22).

Mackworth, Alan K (1977). “Consistency in networks of relations”. In: *Artificial intelligence* 8.1977, pp. 99–118 (cit. on p. 25).

Manhart, Peter (2005). “Reconfiguration - A Problem in Search of Solutions”. In: *Workshop on Configuration In conjunction with the 22nd International Joint Conference on Artificial Intelligence - IJCAI 2005.* Edinburgh, Scotland, UK, pp. 64–67 (cit. on p. 18).

Männistö, Tomi, Timo Soininen, Juha Tiihonen, and Reijo Sulonen (1999). “Framework and Conceptual Model for Reconfiguration”. In: *Configuration Workshop at AAAI 1999* (cit. on p. 18).

Manthey, Norbert, Marijn J.H. Heule, and Armin Biere (2012). “Automated Reencoding of Boolean Formulas”. In: *Haifa Verification Conference.* Vol. 23. Haifa, Israel (cit. on p. 27).

Marques-Silva, Joao P. (May 2008). “Practical applications of Boolean Satisfiability”. In: *9th International Workshop on Discrete Event Systems WODES-2008.* Göteborg, Sweden: IEEE, pp. 74–80. ISBN: 978-1-4244-2592-1. DOI: 10.1109/WODES.2008.4605925 (cit. on p. 39).

Marques-Silva, Joao P. and Inês Lynce (Dec. 2007). “Towards robust CNF encodings of cardinality constraints”. In: *Principles and Practice of Constraint Programming–CP 2007*. Vol. 11. Lecture Notes in Computer Science 3. Springer. Chap. 35, pp. 483–497. DOI: 10.1007/978-3-540-74970-7_35 (cit. on p. 76).

Marques-Silva, Joao P., Inês Lynce, and Sharad Malik (2009). “Conflict-Driven Clause Learning SAT Solvers”. In: *Handbook of Satisfiability*. IOS Press. Chap. 4, pp. 131–153. ISBN: 9781586039295. DOI: 10.3233/978-1-58603-929-5-131 (cit. on p. 38).

Marques-Silva, Joao P. and Karem A. Sakallah (1996). “GRASP-A new search algorithm for satisfiability”. In: *Proceedings of International Conference on Computer Aided Design*. San Jose, California: IEEE Comput. Soc. Press, pp. 220–227. ISBN: 0-8186-7597-7. DOI: 10.1109/ICCAD.1996.569607 (cit. on p. 38).

Mateescu, Robert (2011). *Treewidth in Industrial SAT Benchmarks*. Tech. rep. Cambridge, UK: Microsoft Research. URL: http://research.microsoft.com/pubs/145390/MSR-TR-2011-22.pdf (cit. on p. 52).

Mateescu, Robert and Rina Dechter (2006). “Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs)”. In: *Principles and Practice of Constraint Programming-CP 2006*. Springer, pp. 329–343. DOI: 10.1007/11889205_25 (cit. on pp. 41, 43).

Mateescu, Robert and Rina Dechter (2008). “AND/OR Multi-valued Decision Diagrams for Constraint Networks”. In: *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*. Ed. by Pierpaolo Degano, Rocco De Nicola, and José Meseguer. Vol. 5056. LNCS. Springer, pp. 238–257. ISBN: 978-3-540-68676-7. DOI: 10.1007/978-3-540-68679-8_15 (cit. on p. 43).

Mateescu, Robert, Rina Dechter, and Radu Marinescu (2008). “AND/OR multi-valued decision diagrams (AOMDDs) for graphical models”. In: *Journal of Artificial Intelligence Research* 33, pp. 465–519. DOI: 10.1613/jair.2605 (cit. on p. 43).

McGuinness, Deborah L. (2003). “Configuration”. In: *The Description Logic Handbook*. Ed. by Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter Patel-Schneider. Cambridge University Press. Chap. 12, pp. 388 –405. ISBN: 0-521-78176-0 (cit. on p. 4).

McGuinness, Deborah L. and Jon R. Wright (Sept. 1998). “Conceptual modelling for configuration: A description logic-based approach”. In: *AI EDAM: Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 12.4, pp. 333–344. ISSN: 08900604. DOI: 10.1017/S089006049812406X (cit. on p. 4).

Mendonça, Marcílio (2009). “Efficient Reasoning Techniques for Large Scale Feature Models (PhD thesis)”. PhD thesis. University of Waterloo, Ontario, Canada, p. 170 (cit. on p. 43).

Meseguer, Pedro and Alun D. Preece (July 1995). “Verification and validation of knowledge-based systems with formal specifications”. In: *The Knowledge Engineering Review* 10.04, pp. 331–343. ISSN: 0269-8889. DOI: 10.1017/S0269888900007542 (cit. on p. 12).

Meyer, Marc H. and Alvin P. Lehnerd (1997). *The Power of Product Platforms*. Free Press, p. 288. ISBN: 978-0684825809 (cit. on p. 3).

Miremadi, Sajed (2012). “Symbolic Supervisory Control of Timed Discrete Event Systems (PhD thesis)”. PhD thesis. Chalmers University of Technology, p. 252.

ISBN: 978-91-7385-765-9. URL: http://publications.lib.chalmers.se/publication/164981 (cit. on p. 69).

Miremadi, Sajed, Knut Åkesson, and Bengt Lennartson (2011). "Symbolic computation of reduced guards in supervisory control". In: *IEEE Transactions on Automation Science and Engineering* 8.4, pp. 754–765. ISSN: 1545-5955. DOI: 10.1109/TASE.2011.2146249 (cit. on pp. 56, 67).

Mittal, Sanjay and Felix Frayman (1989). "Towards a generic model of configuration tasks". In: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, pp. 1395–1401 (cit. on pp. 3, 4).

Mohajerani, Sahar, Robi Malik, Simon Ware, and Martin Fabian (June 2011). "On the use of observation equivalence in synthesis abstraction". In: *3rd International Workshop on Dependable Control of Discrete Systems, DCDS-2011*, pp. 84–89. DOI: 10.1109/DCDS.2011.5970323 (cit. on pp. 56, 68).

Montanari, Ugo (Jan. 1974). "Networks of constraints: Fundamental properties and applications to picture processing". In: *Information Sciences* 7, pp. 95–132. ISSN: 00200255. DOI: 10.1016/0020-0255(74)90008-5 (cit. on p. 25).

Moskewicz, Matthew W., Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik (2001). "Chaff: engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. ACM, pp. 530–535. ISBN: 1581132972. DOI: 10.1145/378239.379017 (cit. on p. 38).

Muise, Christian, Sheila A. McIlraith, J. Christopher Beck, and Eric I. Hsu (2012). "Dsharp: Fast d-DNNF Compilation with sharpSAT". In: *Advances in Artificial Intelligence. 25th Canadian Conference on Artificial Intelligence*. Ed. by Leila Kosseim and Diana Inkpen. Toronto, Ontario, Canada: Springer, pp. 356–361. DOI: 10.1007/978-3-642-30353-1_36 (cit. on p. 46).

Narodytska, Nina and Toby Walsh (2007). "Constraint and variable ordering heuristics for compiling configuration problems". In: *Proceedings of the 20th international joint conference on Artifical intelligence*. Hyderabad, India: Morgan Kaufmann Publishers Inc., pp. 149–154 (cit. on p. 45).

Nethercote, Nicholas, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack (Sept. 2007). "MiniZinc: Towards a Standard CP Modelling Language". In: *Thirteenth International Conference on Principles and Practice of Constraint Programming CP-2007*. Ed. by Christian Bessière. Vol. 4741. Lecture Notes in Computer Science. Springer-Verlag, pp. 529–543. DOI: 10.1007/978-3-540-74970-7_38 (cit. on p. 45).

Neubert, Susanne (1993). "Model Construction in MIKE (Model Based and Incremental Knowledge Engineering)". In: *7th European Workshop on Knowledge Acquisition for Knowledge-Based Systems, EKAW '93*. Ed. by N. Aussenac, G. Boy, B. Gaines, M. Linster, J.-G. Ganascia, and Y. Kodratoff. Vol. 723. LNCS. Toulouse and Caylus, France: Springer Berlin Heidelberg, pp. 200–219. ISBN: 978-3-540-47996-3. DOI: 10.1007/3-540-57253-8_55 (cit. on p. 7).

Niedermeier, Rolf (2006). *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, p. 316. ISBN: 978-0-19-856607-6 (cit. on p. 50).

O'Callaghan, Barry, Barry O'Sullivan, and Eugene C. Freuder (2005). "Generating corrective explanations for interactive constraint satisfaction". In: *11th International*

Conference on Principles and Practice of Constraint Programming CP-2005. Ed. by Peter van Beek. Vol. 3709. LNCS. Sitges, Spain: Springer, pp. 445–459. DOI: 10.1007/11564751_34 (cit. on p. 18).

Pan, Guoqiang and Moshe Y. Vardi (2005). "Search vs. Symbolic Techniques in Satisfiability Solving". In: *7th International Conference on Theory and Applications of Satisfiability Testing, SAT 2004, Revised Selected Papers*. Ed. by Holger H. Hoos and David G. Mitchell. Vol. 3542. LNCS. Vancouver, BC, Canada: Springer, pp. 235–250. DOI: 10.1007/11527695_19 (cit. on p. 43).

Pargamin, Bernard (2002). "Vehicle Sales Configuration: the Cluster Tree Approach". In: *Configuration Workshop at ECAI-2002*. July. Lyon, France (cit. on pp. 21, 40, 42).

Pargamin, Bernard (2003). "Extending cluster tree compilation with non-boolean variables in product configuration: a tractable approach to preference-based configuration". In: *Configuration Workshop at IJCAI'03* (cit. on p. 42).

Piller, Frank T. and Christof Stotko (2002). "Mass customization: four approaches to deliver customized products and services with mass production efficiency". In: *IEEE International Engineering Management Conference* 2, pp. 773–778. DOI: 10.1109/IEMC.2002.1038535 (cit. on p. 3).

Pine II, B. Joseph, Bart Victor, and Andrew C. Boynton (1993). "Making Mass Customization Work". In: *Harvard Business Review* (cit. on p. 3).

Pleuss, Andreas, Rick Rabiser, and Goetz Botterweck (2011). "Visualization techniques for application in interactive product configuration". In: *Proceedings of the 15th International Software Product Line Conference on - SPLC '11*. New York, New York, USA: ACM Press, p. 1. ISBN: 9781450307895. DOI: 10.1145/2019136.2019161 (cit. on p. 75).

Pohl, Richard, Kim Lauenroth, and Klaus Pohl (Nov. 2011). "A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models". In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Lawrence, KS: IEEE, pp. 313–322. ISBN: 978-1-4577-1639-3. DOI: 10.1109/ASE.2011.6100068 (cit. on p. 43).

Preece, Alun D., Stéphane Talbot, and Laurence Vignollet (Nov. 1997). "Evaluation of verification tools for knowledge-based systems". In: *International Journal of Human-Computer Studies* 47.5, pp. 629–658. ISSN: 10715819. DOI: 10.1006/ijhc.1997.0152 (cit. on pp. 4, 12, 21).

Prestwich, Steven (2009). "CNF Encodings". In: *Handbook of Satisfiability*. Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications. IOS Press. Chap. 2, pp. 75–97. ISBN: 978-1-58603-929-5 (cit. on p. 26).

Ramadge, Peter J.G. and W. Murray Wonham (1987). "Supervisory Control of a Class of Discrete Event Processes". In: *SIAM Journal on Control and Optimization* 25.1, p. 206. ISSN: 03630129. DOI: 10.1137/0325013 (cit. on pp. 30, 32).

Ramadge, Peter J.G. and W. Murray Wonham (1989). "The control of discrete event systems". In: *Proceedings of the IEEE* 77.1, pp. 81–98. DOI: 10.1109/5.21072 (cit. on pp. 5, 29).

Reiter, Raymond (Apr. 1987). “A theory of diagnosis from first principles”. In: *Artificial Intelligence* 32.1, pp. 57–95. ISSN: 00043702. DOI: 10.1016/0004-3702(87)90062-2 (cit. on p. 18).

Robertson, Neil and P.D Seymour (Feb. 1984). “Graph minors. III. Planar tree-width”. In: *Journal of Combinatorial Theory, Series B* 36.1, pp. 49–64. ISSN: 00958956. DOI: 10.1016/0095-8956(84)90013-3 (cit. on p. 51).

Robinson, John Alan (Jan. 1965). “A Machine-Oriented Logic Based on the Resolution Principle”. In: *Journal of the ACM* 12.1, pp. 23–41. ISSN: 00045411. DOI: 10.1145/321250.321253 (cit. on p. 21).

Rohloff, Kurt and Stephane Lafortune (June 2005). “PSPACE-completeness of Modular Supervisory Control Problems”. In: *Discrete Event Dynamic Systems* 15.2, pp. 145–167. ISSN: 0924-6703. DOI: 10.1007/s10626-004-6210-5 (cit. on p. 30).

Sabharwal, Ashish (Dec. 2009). “SymChaff: exploiting symmetry in a structure-aware satisfiability solver”. In: *Constraints* 14.4, pp. 478–505. DOI: 10.1007/s10601-008-9060-1 (cit. on p. 76).

Sabin, Daniel and Rainer Weigel (1998). “Product configuration frameworks-a survey”. In: *IEEE Intelligent Systems and their Applications,* 13.4, pp. 42–49. DOI: 10.1109/5254.708432 (cit. on p. 3).

Sachenkova, Oxana and Suvash Keshari Thapaliya (2011). *Using CSP solvers for Partial Configuration in Automotive Configuration Problems (Master’s Thesis).* Tech. rep. Göteborg, Sweden: Chalmers University of Technology, p. 42. URL: http://publications.lib.chalmers.se/publication/155929 (cit. on p. 49).

Sakallah, Karem A. (2009). “Symmetry and Satisfiability”. In: *Handbook of Satisfiability.* Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. IOS Press. Chap. 10, pp. 289–338. ISBN: 978-1-58603-929-5. DOI: 10.3233/978-1-58603-929-5-289 (cit. on p. 75).

Samer, Marko and Stefan Szeider (2008). “Backdoor Trees”. In: *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence - AAAI’2008.* Chicago, Illinois, USA: AAAI Press, pp. 363–368 (cit. on p. 51).

Schubert, Monika, Alexander Felfernig, and Florian Reinfrank (2011). “ReAction: Personalized Minimal Repair Adaptations for Customer Requests”. In: *9th International Conference on Flexible Query Answering Systems FQAS-2011.* Ed. by Henning Christiansen, Guy De Tré, Adnan Yazici, Slawomir Zadrozny, Troels Andreasen, and Henrik Legind Larsen. Vol. 7022. LNAI 4. Ghent, Belgium: Springer, pp. 13–24. DOI: 10.1007/978-3-642-24764-4_2 (cit. on p. 18).

Schuh, Günther, Michael Lenders, and J. Arnoscht (Jan. 2009). “Focussing product innovation and fostering economies of scale based on adaptive product platforms”. In: *CIRP Annals - Manufacturing Technology* 58.1, pp. 131–134. ISSN: 00078506. DOI: 10.1016/j.cirp.2009.03.097 (cit. on p. 4).

Simpson, Timothy W., Zahed Siddique, and Jianxin (Roger) Jiao (2006). “Platform-Based Product Family Development: Introduction and Overview”. In: *Product Platform and Product Family Design: Methods and Applications.* Ed. by Timothy W. Simpson, Zahed Siddique, and Jianxin (Roger) Jiao. Springer. Chap. 1, pp. 1–15. ISBN: 978-0-387-25721-1. DOI: 10.1007/0-387-29197-0_1 (cit. on p. 3).

Sinz, Carsten (2005). “Towards an Optimal CNF Encoding of Boolean Cardinality Constraints”. In: *11th International Conference on Principles and Practice of Constraint Programming CP-2005*. Ed. by Peter van Beek. Vol. 3709/2005. Lecture Notes in Computer Science. Sitges, Spain: Springer. Chap. 73, pp. 827–831. DOI: `10.1007/11564751_73` (cit. on pp. 27, 76).

Sinz, Carsten (2006). “Comparing different logic-based representations of automotive parts lists”. In: *ECAI 2006 Workshop on Configuration*, pp. 41–43 (cit. on p. 10).

Sinz, Carsten, Andreas Kaiser, and Wolfgang Küchlin (Aug. 2003). “Formal methods for the validation of automotive product configuration data”. In: *AI EDAM: Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 17.01, pp. 75–97. ISSN: 0890-0604. DOI: `10.1017/S0890060403171065` (cit. on pp. 4, 13, 21, 39, 44).

Soininen, Timo, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen (2001). “Representing configuration knowledge with weight constraint rules”. In: *Spring Symposium on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation at AAAI 2001*. Ed. by Alessandro Provetti and Tran Cao Son. I. AAAI Press, pp. 195–201 (cit. on p. 4).

Stallman, Richard M. and Gerald J. Sussman (Oct. 1977). “Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis”. In: *Artificial Intelligence* 9.2, pp. 135–196. ISSN: 00043702. DOI: `10.1016/0004-3702(77)90029-7` (cit. on p. 25).

Strannegård, Claes, Simon Ulfsbäcker, David Hedqvist, and Tommy Gärling (Oct. 2009). “Reasoning Processes in Propositional Logic”. In: *Journal of Logic, Language and Information* 19.3, pp. 283–314. ISSN: 0925-8531. DOI: `10.1007/s10849-009-9102-0` (cit. on p. 75).

Subbarayan, Sathiamoorthy (2005). “Integrating CSP Decomposition Techniques and BDDs for Compiling Configuration Problems”. In: *Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2005*. Ed. by Roman Barták and Michela Milano. Vol. 3524/2005. LNCS. Prague, Czech Republic: Springer, pp. 351–365. DOI: `10.1007/11493853_26` (cit. on pp. 41, 43, 55).

Subbarayan, Sathiamoorthy, Lucas Bordeaux, and Youssef Hamadi (2007). “Knowledge Compilation Properties of Tree-of-BDDs”. In: *AAAI 2007*, pp. 502–507 (cit. on pp. 40, 41, 43).

Subbarayan, Sathiamoorthy, Rune Møller Jensen, Tarik Hadzic, Henrik Reif Andersen, Jesper Møller, and H. Hulgaard (2004). “Comparing Two Implementations of a Complete and Backtrack-Free Interactive Configurator”. In: *CP-04 Workshop on CSP Techniques with Immediate Application* (cit. on p. 40).

Suwa, Motoi, A Carlisle Scott, and Edward H Shortliffe (1982). “An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System”. In: *AI Magazine* 3.4, pp. 16–21 (cit. on p. 21).

Szeider, Stefan (2003). “Minimal Unsatisfiable Formulas with Bounded Clause-Variable Difference are Fixed-Parameter Tractable”. In: *9th Annual International Conference on Computing and Combinatorics, COCOON 2003*. Ed. by Tandy Warnow and

Binhai Zhu. Vol. 2697. LNCS. Big Sky, MT, USA: Springer, pp. 548–558. DOI: 10.1007/3-540-45071-8 (cit. on p. 75).

Thiel, Steffen and Andreas Hein (July 2002). “Modelling and using product line variability in automotive systems”. In: *IEEE Software* 19.4, pp. 66–72. ISSN: 0740-7459. DOI: 10.1109/MS.2002.1020289 (cit. on pp. 4, 12).

Thüm, Thomas, Don S. Batory, and Christian Kastner (2009). “Reasoning about edits to feature models”. In: *2009 IEEE 31st International Conference on Software Engineering*, pp. 254–264. DOI: 10.1109/ICSE.2009.5070526 (cit. on p. 4).

Tidstam, Anna (2012). “Developing Vehicle Configuration Rules (Lic thesis)”. PhD thesis. Chalmers University of Technology. URL: http://publications.lib.chalmers.se/publication/154850 (cit. on p. 75).

Tidstam, Anna, Lars-Ola Bligård, Fredrik Ekstedt, Alexey Voronov, Knut Åkesson, and Johan Malmqvist (2012). “Development of Industrial Visualization Tools for Validation of Vehicle Configuration Rules”. In: *Proceedings of 9th International Symposium on Tools and Methods of Competitive Engineering (TMCE’12)*, p. 14 (cit. on pp. 14, 16, 75).

Tidstam, Anna and Johan Malmqvist (2010). “Information Modelling for Automotive Configuration”. In: *Proceedings of NordDesign 2010*. Göteborg, Sweden (cit. on p. 10).

Trezentos, Paulo, Inês Lynce, and Arlindo L Oliveira (2010). “Apt-pbo: solving the software dependency problem using pseudo-boolean optimization”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE ’10*. New York, New York, USA: ACM Press, pp. 427–436. ISBN: 9781450301169. DOI: 10.1145/1858996.1859087 (cit. on p. 19).

Tsai, Wei-Tek, R. Vishnuvajjala, and Du Zhang (1999). “Verification and validation of knowledge-based systems”. In: *IEEE Transactions on Knowledge and Data Engineering* 11.1, pp. 202–212. ISSN: 10414347. DOI: 10.1109/69.755629 (cit. on pp. 4, 12, 21).

Tseitin, Gregory S. (1968). “On the complexity of derivation in propositional calculus”. In: *Structures in Constructive Mathematics and Mathematical Logic, Part II, Seminars in Mathematics (translated from Russian)*. Ed. by A.O. Slisenko. Steklov Mathematical Institute, pp. 234–259 (cit. on p. 28).

Uckun, Serdar (2011). *Meta II: formal co-verification of correctness of large-scale cyber-physical systems during design. Volume 1*. Tech. rep. (cit. on p. 21).

Vahidi, Arash, Martin Fabian, and Bengt Lennartson (Oct. 2006). “Efficient supervisory synthesis of large systems”. In: *Control Engineering Practice* 14.10, pp. 1157–1167. DOI: 10.1016/j.conengprac.2006.02.013 (cit. on p. 56).

VDA 4965 (2010). Tech. rep. Verband der Automobilindustrie (German Association of the Automotive Industry) / Strategic Automotive Product Data Standards Industry Group. URL: http://www.vda.de/en/publikationen/publikationen_downloads/detail.php?id=710 (cit. on p. 7).

Veen, Eelco A. van (1992). *Modelling Product Structures by Generic Bills-of-Materials*. New York, NY, USA: Elsevier Science Inc. ISBN: 0444896767 (cit. on p. 14).

Vempaty, Nageshwara Rao (1992). “Solving constraint satisfaction problems using finite state automata”. In: *Proceedings of the tenth national conference on Artificial* *intelligence AAAI-92*. Ed. by Paul Rosenbloom and Peter Szolovits. San Jose, California: AAAI Press, pp. 453–458 (cit. on pp. 42, 56).

Veron, Mathieu, Hélène Fargier, and Michel Aldanondo (1999). "From CSP to configuration problems". In: *AAAI 1999*. AAAI Press, pp. 101–106 (cit. on p. 21).

Voronov, Alexey and Knut Åkesson (2008). "Supervisory control using satisfiability solvers". In: *9th International Workshop on Discrete Event Systems WODES-2008*. IEEE, pp. 81–86. ISBN: 9781424425938. DOI: 10.1109/WODES.2008.4605926 (cit. on p. 50).

Voronov, Alexey, Knut Åkesson, and Fredrik Ekstedt (2011). "Enumeration of valid partial configurations". In: *Configuration Workshop at IJCAI-2011*. Ed. by Kostyantyn Shchekotykhin, Dietmar Jannach, and Markus Zanker. Vol. 755. Barcelona, Spain: CEUR Workshop Proceedings, pp. 25–31 (cit. on pp. 41, 50).

Voronov, Alexey, Knut Åkesson, Anna Tidstam, and Johan Malmqvist (2012). "Verification of Item Usage Rules in Product Configuration". In: *9th International Conference on Product Lifecycle Management*. Montreal, Canada (cit. on pp. 14, 16, 50).

Walsh, Toby (2000). "SAT v CSP". In: *6th International Conference on Principles and Practice of Constraint Programming - CP 2000*. Ed. by Rina Dechter. Vol. 1894. Lecture Notes in Computer Science. Singapore: Springer. Chap. 32, pp. 441–456. DOI: 10.1007/3-540-45349-0_32 (cit. on pp. 27, 28, 43, 76).

Watts, Frank B. (2012). *Engineering Documentation Control Handbook: Configuration Management and Product Lifecycle Management*. 4th ed. Elsevier / William Andrew. ISBN: 978-1-4557-7860-7 (cit. on p. 7).

Williams, Ryan, Carla P. Gomes, and Bart Selman (2003). "Backdoors to typical case complexity". In: *IJCAI 2003* (cit. on p. 50).

Wing, Jeannette M. (Sept. 1990). "A specifier's introduction to formal methods". In: *Computer* 23.9, pp. 8–22. ISSN: 0018-9162. DOI: 10.1109/2.58215 (cit. on p. 21).

Woodcock, Jim, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald (Oct. 2009). "Formal methods: Practice and Experience". In: *ACM Computing Surveys* 41.4, pp. 1–36. ISSN: 03600300. DOI: 10.1145/1592434.1592436 (cit. on p. 21).

Yang, Dong, Ming Dong, and Rui Miao (Aug. 2008). "Development of a product configuration system with an ontology-based approach". In: *Computer-Aided Design* 40.8, pp. 863–878. ISSN: 00104485. DOI: 10.1016/j.cad.2008.05.004 (cit. on p. 4).

Ziyang, Hu (2010). "Analysis and Presentation of Combinatorics in Product Configuration, Master thesis". PhD thesis. Chalmers University of Technology. URL: http://publications.lib.chalmers.se/records/fulltext/128822.pdf (cit. on p. 75).