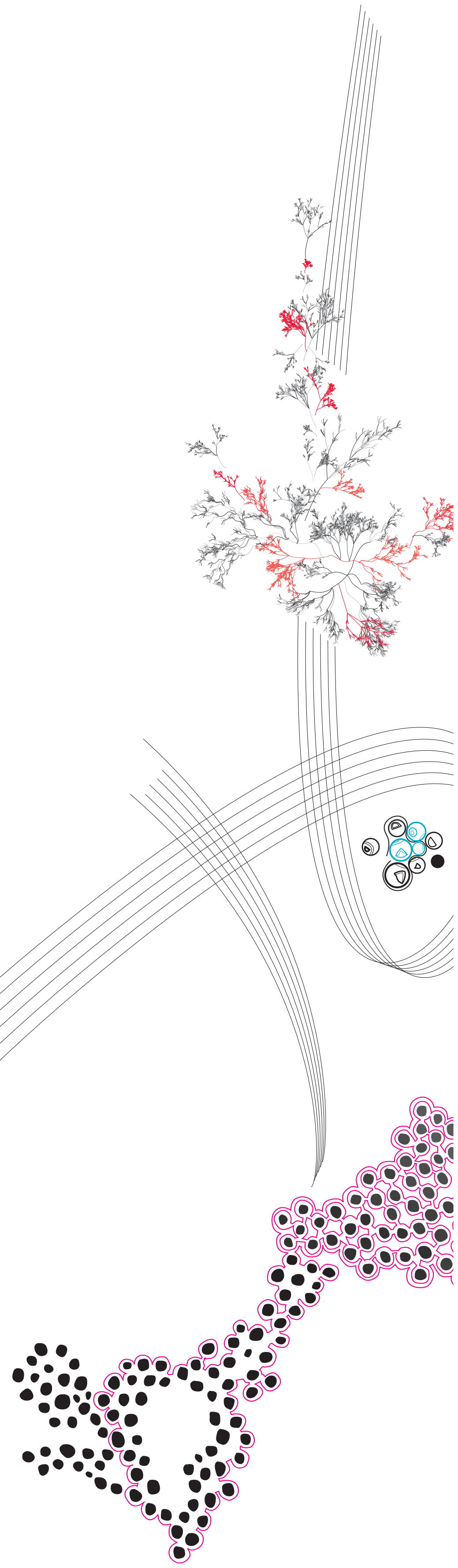
MSc Mechanical Engineering



Binary decision diagram based configurator synthesis for ship locks

T. Scholten
Student number: 2216752
Registration number: DPM2134

Supervisor and Advisors:
Dr. ir. W. Haanstra (UT)
Drs. ing. H.J.W.A.L. Lammeretz (RWS)
Dr. ir. T. Wilschut (Advisor)

July 23, 2024

Department of Mechanical Engineering
Faculty of Engineering Technology,
University of Twente

UNIVERSITY OF TWENTE.

Preface

This document presents the master's thesis titled "Binary decision diagram based configurator synthesis for ship locks". After obtaining my bachelor's degree in Mechanical Engineering at Saxion University of Applied Sciences, I started the Master of Science in Mechanical Engineering, with a specialization in Maintenance Engineering & Operations (MEO) at the University of Twente, Enschede. This research is conducted between September 2023 and July 2024 in collaboration with Rijkswaterstaat.

While looking for a thesis project, I heard that RWS is planning to transition to a Configure to Order (CtO) production strategy. This assignment seemed like an exciting challenge where my academic and professional skills are combined in a socially important project. During this thesis, I gained many new insights, e.g. into locks, CtO and learned programming in Python. Furthermore, I learned the importance of defining the projects objective and scope beforehand and during projects I came across many unforeseen obstacles and learned how to solve them. Sometimes, taking a step back is necessary to move forward. This thesis has taught me important lessons both professionally and personally.

I would like to thank my supervisors, Dr. ir. W. Haanstra (UT), Drs. ing. H.J.W.A.L. Lammeretz (RWS), and Dr. ir. T. Wilschut (Advisor), for their support during this thesis. We had many productive discussions that provided me with valuable insights for this thesis. I would also like to thank Quootz B.V. for providing their product configuration software throughout the duration of this thesis.

Lastly, I want to thank my family and friends for their support during my thesis period. I hope you enjoy reading this.

Tom Scholten
Apeldoorn, July 23, 2024

Abstract

Rijkswaterstaat (RWS), the executive branch of the Dutch Ministry of Infrastructure and Water Management, has a significant challenge in renovating and replacing navigation locks. Due to the labour intensity of the current Engineer to Order (EtO) production strategy, RWS decided to transition towards a Configure to Order (CtO) production strategy. The CtO strategy is supported by a product platform. The components of the product platform have restrictions on how they may be combined and have to align with the project requirements. Therefore, the selection of valid configurations using components from the product platform is facilitated by a product configurator. A configurator simplifies the practical application of a product platform.

Product configurators are widely used in engineering-oriented companies to configure products and have resulted in many positive effects. However, the development of a configurator often turns out to be more challenging and labor-intensive than initially expected. Configurators for complex products, such as locks, have many components and restrictions, and the number of different configurations increases exponentially with the components in the product platform. Furthermore, the configurators are manually developed, verified, and validated, which are error-prone and labor-intensive especially for complex products.

Often product configurators are used to configure new products, however the lock configurator should be suitable for replacement and renovation of an existing lock. In the latter case, the current situation and the scope for the different components, e.g. renovate, replace or retain, influences what may be chosen in the new configuration. Furthermore, the design of a lock is largely determined by the demands of the location in the waterway network and by environmental conditions at that location. Location specific requirements (LSR) impose restrictions on the desired lock configuration.

Given the adverse effects of manual development and the lack of a method to automatically develop a configurator based on the product platform and additional requirements, RWS wants to investigate the possibility of automatically generating the configurator. This would help avoid human errors and make it easier to create valid configurations and how to deal with the challenges posed by the configuration of renovation projects.

In current research, a method is proposed for the automatic generation of a product configurator, from which corresponding solution space can be defined. From the product platform, LSR and design rules, corresponding product configurator can be synthesised using Binary Decision Diagrams, which guarantees to produce only valid configurations. Moreover, this configurator is used to create configurations for both replacement and renovation projects. Furthermore, the design rules for renovation projects are automatically calculated. This way, RWS users no longer have to manually develop and verify the configurator, but only to specify the design rules and validate that specified design rules are correct. The proposed method is demonstrated by synthesising a configurator for a complex lock module based on the product platform and project requirements. The method enhances efficiency, minimizes error sensitivity, reduces labor intensity, facilitates ease of maintenance and adaptability, all while guaranteeing the production of valid configurations. Furthermore, the method is generic and can therefore also be applied for other companies that want to synthesis a configurator.

Contents

List of Figures

List of Tables

Glossary

(Product) Configurator	Software-based expert system that assists users in creating product specifications by limiting how various components and properties can be combined [41][13].
Binary decision diagram (BDD)	A binary decision diagram (BDD) is a data structure used to represent a Boolean function by a finite Directed Acyclic Graph (DAG) [10].
Brownfield project	Project for the partial replacement and/or renovation of an existing lock.
Component	A part of a lock for which a variant can be chosen.
Component scope	In a brownfield project, the scope for each component must be specified, indicating whether the component is to be replaced or renovated.
Configuration	The set of selected variants for a specific project.
Configurator rule	Rule used in the configurator in the form IF *condition* THEN *consequence*. This is either a question-visible-, answer-visible- and calculation- rule.
Configurator software	Software that is used to execute our configurator. For this, the Merkato software is used.
Configure to order (CtO)	Production strategy in which products are configured according to each customer's specifications.
Current situation	Present component variants according to the product platform in a lock.
Design rules	Excluded-, enforced combinations and precedence rules.
Enforced combination	A combination of two or more (component) variants enforced to select if all (component) variants, in the combination, except one is selected.

Engineer to order (EtO)	Production strategy in which products are designed according to each customer's specifications.
Excluded combination	A combination of two or more (component) variants that cannot be selected together.
Greenfield project	Project for the construction of a new lock or for the complete replacement of a existing lock.
GRIP	The project management system used by RWS in which the product platform is implemented.
Incompatible combination	A combination of two or more variants that may not be selected together.
Location specific requirements (LSR)	Requirements originating from the lock's location that influence the lock's design.
Mutually compatible	A set of (component) variants that may be selected together.
Product platform	Defined by [21] as: "A set of common components, modules, or parts from which a stream of derivative products can be efficiently developed and launched".
Project specific requirements (PSR)	Requirements specifically for a single project, such as the LSR, project type, current situation and component scope
Project type (PT)	The type of the project is either greenfield or brownfield.
Rule-based product configurator	Product configurator paradigm that uses design rules of the form IF *condition* THEN *consequence*.
Solution space	All valid configurations produced by a configurator.
Synthesis	Algebraic calculation.
Valid configuration	Configuration that complies with all requirements.
Variant	A technical solution for a component.

Chapter 1

Introduction

The Netherlands has one of the busiest waterway networks in the world [29]. Vital assets of the waterway infrastructure are navigation locks. A navigation lock is a structure in the waterway to regulate the water flow and to enable safe navigation passage between different levels in the waterways [27]. Navigation locks in Dutch waterways are complex systems consisting of many components (and component interfaces) uniquely made to meet location specific requirements (LSR). Figure 1.1 shows a lock complex with two chambers located in Eefde, the Netherlands. The left chamber was constructed after the right chamber. A segment gate and a miter gate are utilized for the upper and lower gate of the left chamber, respectively, while lift gates are used for the right chamber.

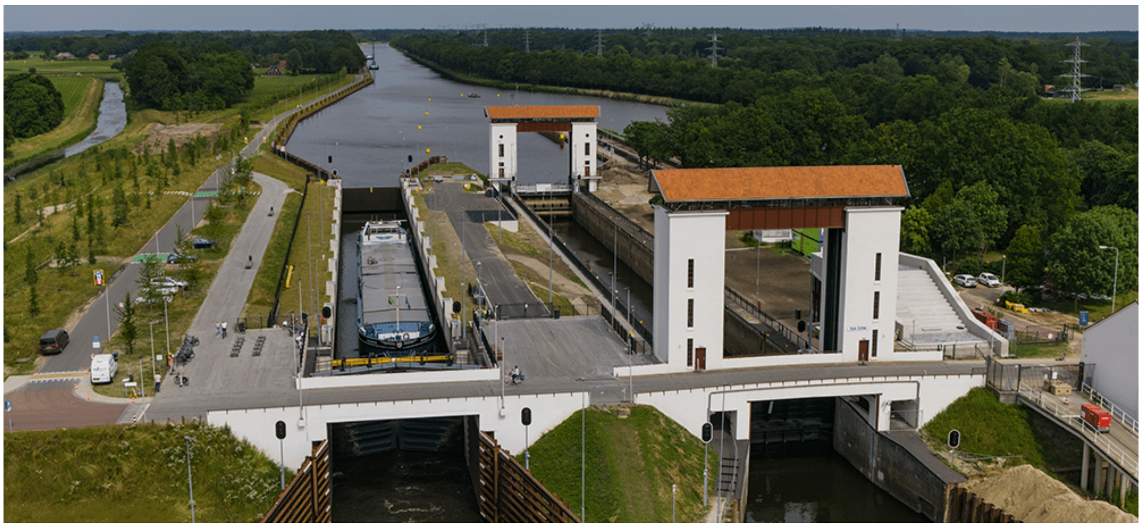


Figure 1.1: Lock complex with two chambers in Eefde, the Netherlands

The 128 different navigation locks located in the main waterways of the Netherlands are owned and operated by Rijkswaterstaat (RWS)[20]. RWS is the executive branch of the Dutch Ministry of Infrastructure and Water Management and responsible for the design, construction, management and maintenance of the main infrastructure facilities in the Netherlands [28].

RWS faces a significant challenge. In the coming two decades, over 50 navigation locks have to be thoroughly renovated or replaced, since they have reached their end-of-life, no longer meet modern-day safety standards, or have insufficient capacity to keep up with growing waterborne transportation [38] [20].

The lock's requirement specification is elaborated by RWS, which outsources the design and construction of locks to external parties. To this day, locks are designed according to an

Engineer-to-Order (EtO) product realisation strategy, where navigation locks are individually tailored to meet project specific requirements (PSR) in which local stakeholders and engineers use their personal preferences in designs. As a result, current requirements specification is not standardised and has led to a large variety of locks in the product portfolio [18]. The reliance on local specialized knowledge for operating and maintaining these locks, along with the need for numerous costly and unique spare parts, has a detrimental effect on the reliability, availability, maintainability, safety (RAMS), and Life Cycle Costing (LCC) of the locks [37]. Furthermore, EtO production is highly labor-intensive and requires a considerable amount of expertise from the project team resulting in long realisation times. Simultaneously, the civil engineering and construction sector is grappling with diminishing market competitiveness, limited innovation, slow productivity growth, and a shortage of well-trained technical personnel [39]. In view of the scale of the task, concerns about the increasing variety and labour intensity, RWS wants to reconsider the product realisation strategy.

Companies across other branches of industry are experiencing similar negative effects due to the EtO product realisation strategy [19]. Often, the high degree of customization offered by this strategy is unnecessary. By increasing the degree of standardization, companies can reduce product variety and negative side effects. A more standardized and often used alternative product realisation strategy is Configure-to-order (CtO). CtO utilises predefined standard modules and components to configure a realisable product that meets customer requirements [37]. Due to the use of standard modules and components, CtO offers a higher standardisation as well as a lower degree of customization compared to EtO.

To address these challenges, RWS founded the MultiWaterWerk (MWW) project in 2014. Within the MWW project, methods and tools are being developed to efficiently carry out the replacement and renovation of locks and to improve the performance of the locks. For the replacement and renovation of locks in the next to decades, MWW has budgeted several billion euros. The MWW project primarily comprises two research lines: civil-mechanical and industrial automation[39]. The civil-mechanical research line aims to develop methods for standardisation and modularisation of the navigation lock portfolio in order to increase RAMS and decrease LCC of future locks. As part of the MWW project, it is agreed that these goals can be achieved by shifting from an EtO to a Configure to Order (CtO) product realisation strategy [20]. Furthermore, as part of the MWW project, RWS decided to utilize this strategy for bridges and tunnels as well.

A product platform can provide the standardised modules and components for the CtO production strategy. A product platform is formally defined by [21] as: "A set of common components, modules, or parts from which a stream of derivative products can be efficiently developed and launched". The MWW project started several years ago, including the development of a product platform for locks for which the first concept has already been built. The product platform has a hierarchical structure comprising components and their (component) variants, for which requirements specifications are made. A requirements specification for a lock can be configured by selecting variants for necessary components of the lock. The current implementation of the product platform lacks restrictions on the selection of variants. As a result, users have the freedom to select any combination of variants from the product platform. Hence, this can lead to invalid configurations, as further explained in Chapter 2.

The selection of valid configurations from a product platform can be facilitated by a product configurator. A product configurator is a software-based expert system that assists users in creating product specifications by limiting how various components and properties can be combined [41][13]. The configurator asks the necessary product-related questions, accepts only valid answers, and thus supports the user in selecting a valid configuration from the product platform during the product configuration process. A product configurator uses configurator rules to determine when what product-related questions are necessary and restricts how answers can be combined.

Components of the product platform relate to those questions, while variants of the product platform relate to the answers. Configurator rules can be derived from design rules, which impose restrictions on the need of components and the use of variants, such that configurations meet PSR as well as existing guidelines and manuals on lock design. If adhered to all design rules, a configuration is valid. This configuration can then be utilized to generate a standardized requirements specification. In conclusion, a configurator assist RWS engineers in the creation of valid lock configurations from the product platform.

1.1 Problem description

Product configurators are widely used in engineering-oriented companies, including the automotive industry, and have resulted in many positive effects. That is, reducing lead time in specification processes, ensuring on-time delivery of specifications, lowering resource consumption for creating specifications, enhancing the quality of specifications, and optimizing products and services [2] [17]. Together with performance improvements, configurators also bring many changes and difficulties along [42]. Most companies struggle with designing, developing, and maintaining product configurators due to a shortage of IT system designers and poor communication between IT system designers and product designers[42]. The development of a configurator often turns out to be more challenging and labor-intensive than initially expected [14].

Since the early '90s, product configurators have been a popular subject of study in both academia and industry. This has resulted in many publications on theoretical aspects such as configuration knowledge and design methods. However, only a small number of articles focus on the practical applications of product configurators [42].

Traditionally, configurators are manually developed, verified, and validated. For complex products, difficult configurator rules are developed manually. Additionally, it must be verified and validated that only valid configurations can be made, which is accomplished through manual testing of the configurator. Manual development, verification, and validation of the product configurator are error-prone and labor-intensive [36].

In literature, configurations where the user is assisted by the configurator in making choices are referred to as the 'Interactive Configuration Problem'. After each user interaction, a reassessment of permissible choices is necessary [12]. The Interactive Configuration Problem is often modeled in the literature as a Constraint Satisfaction Problem (CSP) [34] [12]. A CSP is defined as a mathematical problem by a set of variables, each with a specific domain of values, and a set of constraints specifying allowable combinations of values [34] as further explained in Chapter 3. Solving a CSP provides all possible variable assignments that meet the specified constraints. A CSP is generally NP-complete, which implies that solving it may require exponential time as the number of variables increases, making such problems impractically difficult with larger input sizes [7].

Configurators for complex products, such as locks, have many questions, answers and configurator rules. Single design rules can relatively easy be formulated from handbooks and put together, however the derivation of configurator rules from design rules can become highly complex due to many restrictions and design rules that use overlapping answers. The number of different configurations increase exponentially with the number of questions and corresponding answers, resulting millions or even billions of different configurations. However, these configurations might contain combinations of answers that may not be chosen together as they conflict with design rules, which leads to invalid configurations. All valid configurations produced by a configurator are referred to as the solution space.

A CSP can be represented using a Binary Decision Diagram (BDD). BDD's are data structures used to efficiently represent and manipulate Boolean variables and functions as further explained in Chapter 4. Solving the BDD for complex products, like locks, can potentially result in memory explosion[36] [15]. Furthermore, the solution space only contains final configurations but does directly provide configuration support for the user.

Often product configurators are used to configure new products, which is in this case the same as replacing a lock completely. However the lock configurator should not only be suitable for replacement (referred to as greenfield), but also for renovation of the existing lock (referred to as brownfield). In case of brownfield projects, the entire current situation influences what can be chosen [34], and for RWS, each component has its own component scope. Furthermore, brownfield projects may include non-standardized components that RWS aims to phase out economically during replacements. This raises the question of how to model constraints for brownfield projects considering the current situation and component scope. Other research indicates that renovation projects in the civil engineering sector is challenging [34].

The configurator to be developed for RWS must be practically applicable for designing locks. Product knowledge is represented by the hierarchical product platform, LSR and design rules. LSR are used to describe the situation for which the lock is used and design rules limit how LSR and variants can be combined. Based on this knowledge a configurator must be developed. This configurator should be produce valid configurations, easy to manage and maintain and suitable for both greenfield and brownfield projects. However, literature lacks a generic method that can be used for this purpose.

Given the adverse effects of manual development and lack of a method to automatically develop a configurator given the product platform and additional requirements, RWS wants to investigate the possibility of automatically generating the configurator to avoid human errors and make it easy to create valid configurations and how to deal with challenges posed by the configuration of brownfield projects.

1.2 Research objective

As a result of the problem statement, an alternative method for development of a product configurator is examined. In this thesis, a method is proposed for the automatic generation of a product configurator, from which corresponding solution space can be defined.

Automatic generation of the product configurator and thus configurator rules ensure configurator rules are correctly developed, which means engineers no longer have to manually develop and verify configurator rules. The goal of this method is to enhance efficiency, minimize error sensitivity, reduce labor intensity, facilitate ease of maintenance and adaptability, all while guaranteeing the production of valid configurations. If the product platform or requirements change, then the method should be reusable to regenerate the adjusted configurator.

The method provides not only a functional configurator, but also ensures its user friendliness. To achieve this, we aim to integrate the generated configurator into a commercial product configurator equipped with the necessary features for user-friendliness. The commercial product configurator used in this research is Merkato.

This research answers the following research questions:

1. How can the configurator solution space be defined?
2. How can a configurator be developed to guarantee the production of valid configurations?
3. How can a configurator be used for both greenfield and brownfield projects?
4. What approach can be utilized to facilitate configurator management and maintenance?

1.3 Contribution

General methodology development
The main contribution of this thesis is the development of a generic method, applicable for generating a configurator for any product. Therefore also other companies can use and benefit from this developed method. Furthermore, this method can be used for both greenfield and brownfield product configurators, which is referred to as the project type (PT).

The method synthesises the configurator rules, consisting of question-visible-, answer-visible- and calculate- rules, based on a set of hierarchical questions with discrete answers and design rules. This research is limited to the use of closed questions, such that each question has a discrete set of answers of which to choose from. BDD are used for algebraic calculation of the answer-visible-rules. By creating variables for answers and Boolean conditions for design rules, configurator rules can be synthesised which guarantee only valid configurations can be made. To guide the user through the questions, question-visible-rules are derived from the product platform hierarchy and precedence constraints. Calculate rules are used for invisible questions to make sure every question has an answer.

Application to RWS locks
In this project, the use of the method is demonstrated by creating a lock configurator for RWS, as illustrated by the process scheme in Figure 1.2. The numbers in this scheme refer to the chapters in which the steps are discussed. First a configurator data structure is generated from the product platform, LSR and design rules. This is then used in the main method to synthesize question-visible-, answer-visible- and calculate- rules. These rules are then used to generate a configurator template which can be imported into the Merkato product configurator. RWS users can use this configurator to create lock configurations. Based on the chosen answers in a configuration and corresponding requirements specifications from the product platform, the requirements specification for a specific project, can be generated. If the configurator needs any changes, this can be done to the input of the configurator data structure and regenerate the configurator. The validity and practical applicability of the developed method is demonstrated by conducting a case study on a RWS lock. The implementation of the method is programmed in Python.

1.4 Outline

This thesis is organized as follows: Chapter 2 provides an overview of the specific background of RWS. Chapter 3 describes theories found in the literature, which serve as the foundation for the method developed in Chapter 4 for generating a configurator. The application of the method is demonstrated through the development of a configurator for a complex lock module in Chapter 5.

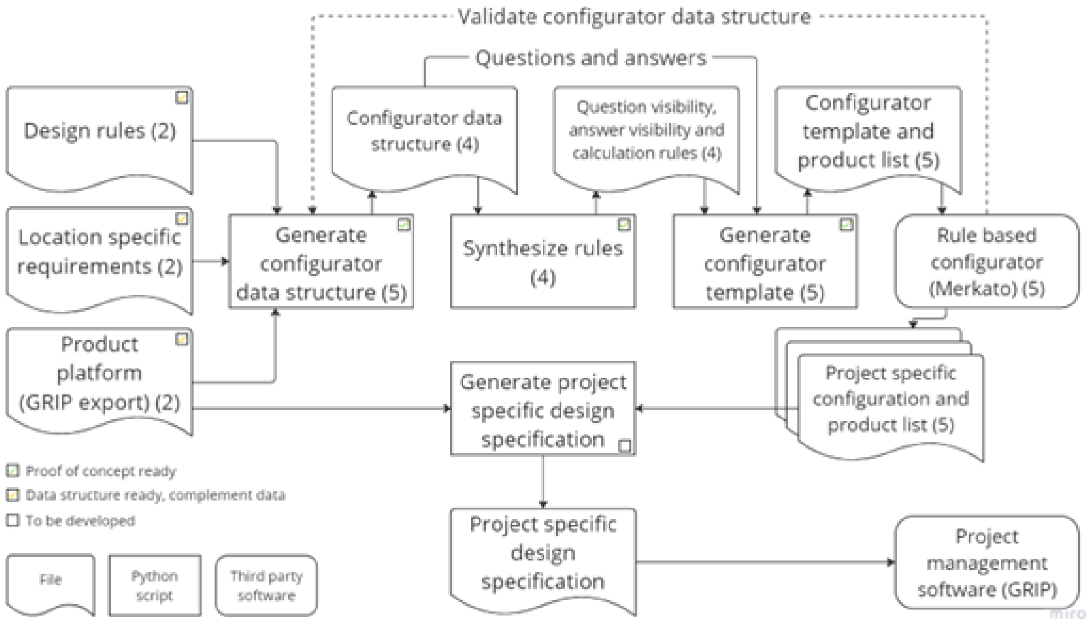

Validate configurator data structure
Questions and answers
Design rules (2)
Configurator data structure (4)
Question visibility, answer visibility and calculation rules (4)
Configurator template and product list (5)
Location specific requirements (2)
Generate configurator data structure (5)
Synthesize rules (4)
Generate configurator template (5)
Rule based configurator (Merkato) (5)
Product platform (GRIP export) (2)
Generate project specific design specification
Project specific configuration and product list (5)
Proof of concept ready
Data structure ready, complement data
To be developed
File
Python script
Third party software
Project specific design specification
Project management software (GRIP)
miro

Figure 1.2: Process scheme illustrating the application of the method

Chapter 2

Product platform navigation lock

A product platform has been developed to support both the replacement and renovation of navigation locks. This product platform will be implemented through a product configurator. This configurator should only produce valid configurations. The validity of a configuration depends on requirements. This chapter describes the components of a lock, the product platform, the requirements of valid configurations and the requirements of the product configurator.

2.1 Lock composition

Previous research has decomposed a generic lock into components. A component is a part of a lock and a variant is the technical solution for a component. For example, the component ”Lock gate” has variants ”Miter gate”, ”Double miter gate” and ”Lift gate”. Some components are always present in every lock, while other components are optional.

In the research by [24] and [18], a schematic top view of a lock is presented, as shown in Figure 2.1. The lock in this figure has two lock heads, as indicated by the dashed lines. The numbers in this figure indicate the main components of the lock and a description of the components is given below. The main components are also the main determinants for the civil-mechanical lock design. Moreover, the components within the lock head construction have the most complex design rules. Therefore, the examples will be based on these components referred to as lock head.

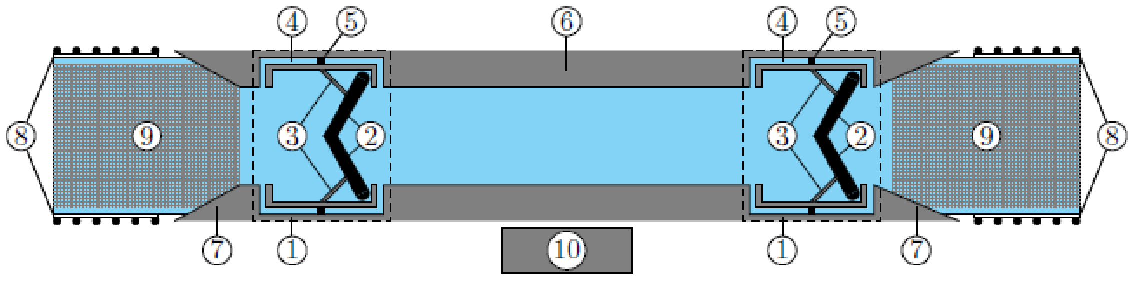


Figure 2.1: Top view of a lock with two lock heads (dashed) [24][18]

According to [18], the lock consists of the following elements:

1. Head construction: provides support to components within;
2. Gate: retains water when closed and allows ships to pass when open;
3. Gate actuator: opens and closes the gate;
4. Leveling system: levels water inside the lock chamber. This component is optional, since water can also be leveled by partly opening the gate;

5. Leveling actuator: opens and closes the leveling system valves. This component is optional for the same reason as leveling system;
6. Chamber construction: isolates a part of the waterway between the gates where the water level can be raised and lowered while containing ships that need to pass the lock;
7. Leading jetty: guides a ship into the lock;
8. Positioning area: positions a ship to enter the lock and provides support for ships;
9. Soil protection: protects the soil outside the lock from propeller turbulence and water outflow;
10. Control system: controls the correct and safe dynamic behaviour of the lock.

2.2 Product platform

In the research by [18], a lock product platform has been developed based on a lock family analysis of the current lock portfolio. The product platform has a hierarchical structure consisting of components and (component) variants. Each component has one or more variants, and in turn a variant consists of one or more (possibly optional) components. As a result, components and variants alternate with each other as schematically depicted in the model structure in Figure 2.2.

The product platform is based on the physical system decomposition of locks. Figure 2.3 shows part of the product platform. In this figure, variants are denoted by the prefix "VARIANT" and optional components are denoted by the prefix "OPTIE".

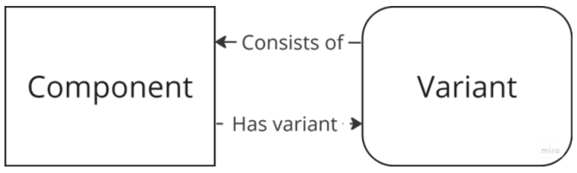


Figure 2.2: Relationships between the model elements component and (component) variant [40]

The product platform contains most common variants and their components used in the current lock portfolio. Special variants are not included. RWS wants to phase out some variants, like "Heatflow ice prevention system" whenever it is replaced.

The product platform supports a CtO. The hierarchy of the product platform dictates for which component a variant should be selected and when. Starting from the root of the product platform hierarchy, a lock can be configured by selecting a variant for each component. Each selected variant consists of components for which again a variant must be chosen. This already limits the use of certain combinations of variants, e.g. variants "Wood" or "Steel" may not be chosen in combination with the variant "Lift gate".

However, there are additional combinations of variants that may not be chosen together due to the technical characteristics of the variants. These combinations that may not be selected together are referred to as incompatible. For example, if for component "Lock gate" variant "Single miter gate" is chosen, then for component "Gate actuator" variant "Push pull gate actuator" must be chosen. An incompatibility can occur between two or more variants. It is important that the selected variants in a configuration are mutually compatible. The compatibility can be derived from, among others, the Landelijke Brug- en Sluisstandaard (LBS). Furthermore, a configuration must also satisfy LSR and PT in order to be valid, as further explained in the next paragraphs.

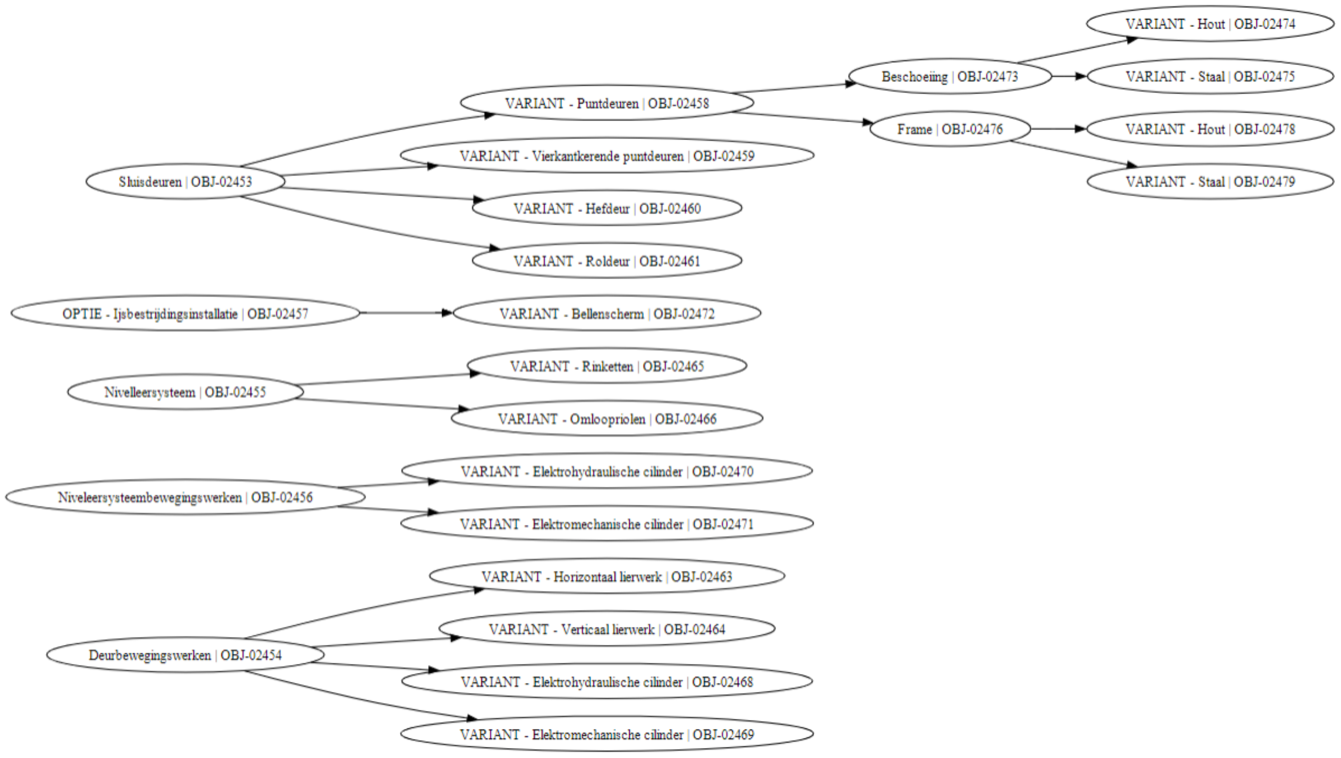

Sluisdeuren | OBJ-02453
VARIANT - Puntdeuren | OBJ-02458
Beschoeiing | OBJ-02473
VARIANT - Hout | OBJ-02474
VARIANT - Staal | OBJ-02475
Frame | OBJ-02476
VARIANT - Hout | OBJ-02478
VARIANT - Staal | OBJ-02479
VARIANT - Vierkantkerende puntdeuren | OBJ-02459
VARIANT - Hefdeur | OBJ-02460
VARIANT - Roldeur | OBJ-02461
OPTIE - Ijsbestrijdingsinstallatie | OBJ-02457
VARIANT - Bellenscherm | OBJ-02472
Nivelleersysteem | OBJ-02455
VARIANT - Rinketten | OBJ-02465
VARIANT - Omloopriolen | OBJ-02466
Niveleersysteembewegingswerken | OBJ-02456
VARIANT - Elektrohydraulische cilinder | OBJ-02470
VARIANT - Elektromechanische cilinder | OBJ-02471
Deurbewegingswerken | OBJ-02454
VARIANT - Horizontaal lierwerk | OBJ-02463
VARIANT - Verticaal lierwerk | OBJ-02464
VARIANT - Elektrohydraulische cilinder | OBJ-02468
VARIANT - Elektromechanische cilinder | OBJ-02469

Figure 2.3: Example product platform for the lock head

The current product platform is implemented in GRIP, the project management system used by RWS. GRIP does not support the necessary restrictions on compatibility, LSR and PT, resulting in invalid configurations. Using a dedicated product configurator, RWS can make the right selections for a projects in GRIP.

2.3 Location specific requirements

The LSR of navigation locks in Dutch waterways are largely determined by the demands of the location in the waterway network and by environmental conditions at that location. Location specific requirements impose restrictions on the desired lock configuration. This implies that all selected variants must adhere to the specified LSR. A design rule for for the LSR contains one or more LSR and one or more variants. For example, for component "Lock gate", variant "Single miter gate" may not be selected if the "Water retention" is "Bi-directional". A concept of the constraint requirements for lock design is displayed in Table 2.1. The exact LSR that RWS needs can be determined later. The LSR can be derived from, among others, the LBS.

2.4 Project type requirements

The configurator will be used for locks that need to be replaced or renovated, referred to as greenfield and brownfield respectively. In a greenfield project, an existing lock is completely demolished and rebuilt. Brownfield projects consist of replacement and renovation work on an existing lock.

An existing lock can often be described as a set of selected variants of a product platform. If not, it means that it contains special variants that are not included in the product platform. It is highly likely that these variants need to be replaced with a better alternative, within the framework of standardization.

Table 2.1: Location specific requirements [18]

Questions	Answers
Waterway class	CEMT 0
	CEMT I
	CEMT II
	CEMT III
	CEMT IV
	CEMT Va
	CEMT Vb
	CEMT VIa
	CEMT Via
Lock head width	≤ 8 m
	> 8 and ≤ 12 m
	> 12 and ≤ 16 m
	> 16 and ≤ 20 m
	> 20 and ≤ 24 m
	> 24 m
Water retention	Mono-directional
	Bi-directional
Water level difference	≤ 3 m
	> 3 and ≤ 6 m
	> 6 and ≤ 9 m
	> 9 m
Water level variation	≤ 1 m
	> 1 m
Operation and control	Local
	Remote

The component scope determines which variants of the lock need to be replaced or renovated. For greenfield locks, all variants are replaced. For brownfield projects, some variants need to be replaced or renovated. If a variant is not replaced, the current variant is retained. If the variant is replaced, it is possible to choose a different variant. If a new configuration is made for a brownfield lock, this new configuration must not deviate from the current situation for the variants that are not being replaced. Brownfield projects thus have additional restrictions on what can be chosen from the product platform, based on the current situation and component scope. For example, if in the current situation, for the component "Lock gate", the variant "Single miter gate" is selected and the component scope for the component "Lock gate" is either "Renovate" or "Retain", then for the new situation, the same variant "Single miter gate" must be chosen. However, if the component scope for that component is "Replace", then the variant may deviate from the current variant.

2.5 Product configurator

The developed configurator can be used by RWS users to configure valid lock configurations based on the LSR and PT. The RWS users are guided via an interactive process and gives answers to questions to find a valid configuration.

The development of a method for automatic generating a product configurator is done in Python and extra functionality is achieved by integrating the configurator into professional product configurator software.

There are several product configurator paradigms, namely rule-based, model-based, and case-based [4]. Rule-based systems use configurator rules of the form IF *condition* THEN *consequence*. Model-based systems use a system model that contains decomposable entities and interactions between their elements. Case-based systems use previous configurations and adapt them to current requirements. In this project, the choice is made to work with a rule-based configurator because it is flexible, transparent, and utilizes IF *condition* THEN *consequence* as proposed in previous research [18]. Additionally, model-independent questions can be easily added.

The commercial product configurator software used in this research is Merkato. However another rule based and non sequential configurator could also be used to provide extra functionality. Merkato is developed by Quootz B.V., who claims to be the market leader in product configurator software, with their software being trusted by over 100 companies [5]. Figure 2.4 shows the default home screen of Merkato.

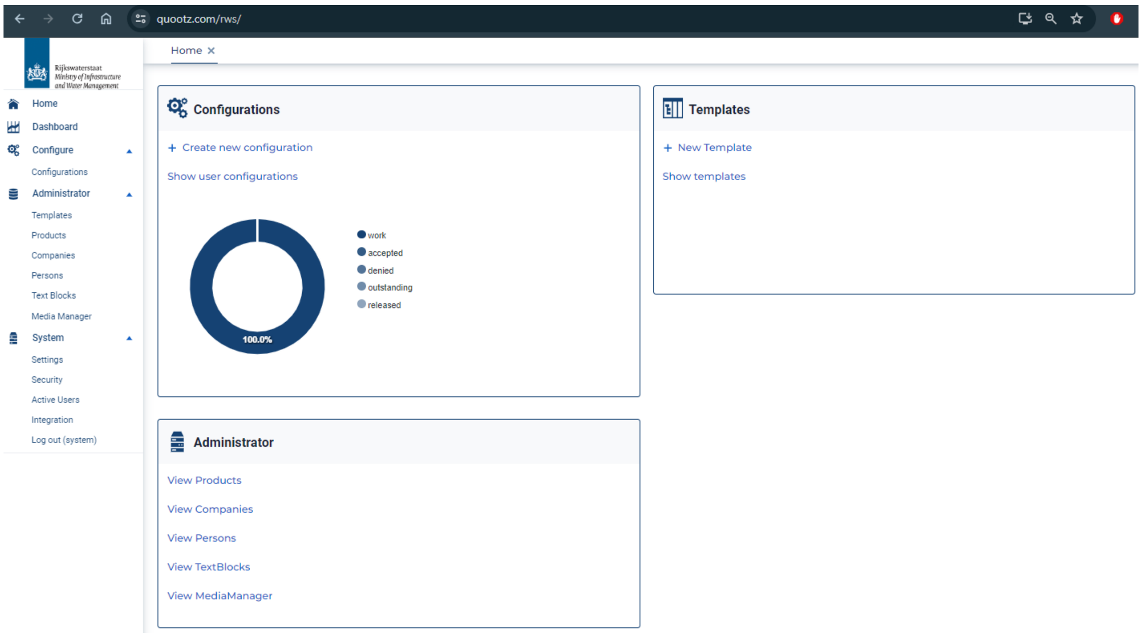


Figure 2.4: Merkato home screen

Questions, answers and configurator rules are referred to as fields, options/values and decision tables in Merkato respectively. Information about how a configuration should behave is stored in a template.

In Merkato, templates can be manually developed via a graphical user interface. Merkato offers many functionalities for template development. Figure 2.5 shows the Merkato template view. In this view, a page named ”Page 1” has been created, containing a section named ”Section 1”. Within ”Section 1”, two fields, each with two options, have been added. Fields can have various input types, such as text, number, and select. However, since only discrete values are of interest, the select input type is used. Two options are added to each of the select fields. Furthermore, a decision table is added to the second field and is shown on the right side of the figure.

A decision table consists of configurator rules in the form IF *condition* THEN *consequence*. Each row represents a new IF *condition* THEN *consequence*. The right column in a decision table contains the consequence, and one or more columns on the left contain the conditions. If a condition is true, then the consequence is executed. The table in Figure 2.5 can be read as: IF *field_1 is equal to option_1_field_1* THEN *option_1_field_2 is visible* and IF *field_1 is equal to option_2_field_1* THEN *option_2_field_2 is visible*. Merkato offers different kinds of consequences, as shown below:

- Calculate tables: field value will be set to consequence value;
- Visible tables: shows or hides a field;
- Enabled tables: enables or disables a field;
- Product tables: used to put products on the quote list, which is a list of the products used in that particular configuration;
- Option visible tables: used to show or hide options;
- Option enabled tables: used to enable or disable options.

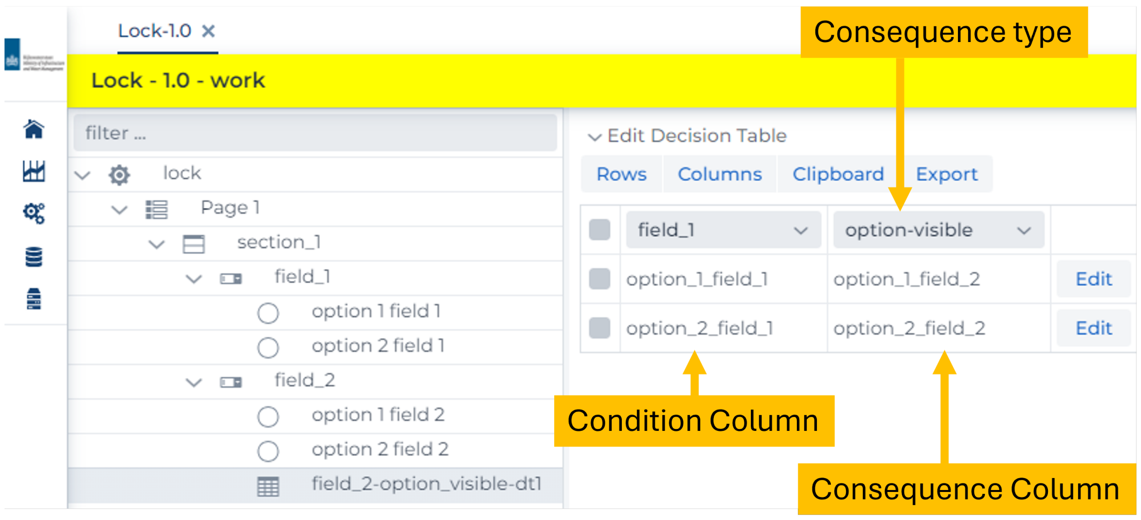


Figure 2.5: Merkato template view

From a template, configurations can be made. The setup of the template determines how the configuration should behave. Figure 2.6 shows the corresponding configuration to the template of Figure 2.5. It can be observed that "option_1_field_1" is chosen for "field_1", and as a result, only option "option_1_field_2" is visible for "field_2", as specified in the option visibility table.

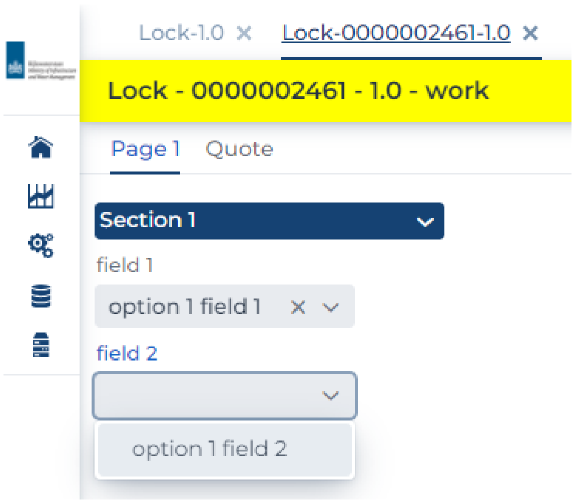


Figure 2.6: Merkato configuration view

Additionally Merkato offers a user-friendly interface, web-based functionality, easy and secure role-based access, version control, bill of material generation, document generation, and other functionality that RWS need already or might need in the future.

2.6 Requirements specification

As explained in the introduction, RWS only elaborates on the requirements specification for the replacement or renovation of locks. The requirements specification is a structured set of requirements that contractors, to whom the project is outsourced, must comply with. The variants in the product platform are therefore not tangible but rather a sub-requirement specification for that variant. Based on the chosen variants, it can be determined which sub-requirement specifications are needed in the project's requirements specification. In the case of brownfield projects, additional requirements are needed depending on the current situation and component scope. [33]

Standardized sub-requirements specifications can be derived from the Basisspecificatie (BS) Schutsluis, LBS, and Richtlijn Ontwerp and Kunstwerken (ROK), and are listed in GRIP. These sub-requirements specifications may encompass functional requirements or specifications regarding the minimal performance of the lock, such as RAMS, as well as LCC [18] [38].

Chapter 3

Theoretical background

The literature has been briefly scanned for related work. This chapter presents the most important works that provide the motivation and theoretical basis for the proposed method discussed in Chapter 4.

3.1 Solution space definition

Constrained programming is widely used in industry to model and solve decision problems [35][11]. A configurable product can be modeled by a Constrained Satisfaction Problem (CSP) of which the solutions are valid configurations [35].

A (finite domain) CSP can be expressed by a triplet $P = X, D, C$, where:

- $X = \{X_1, X_2, ..., X_n\}$ is a set of n variables;
- $D = \{D_1, D_2, ..., D_n\}$ is a set of n corresponding finite domains. Every X_i can take a value of corresponding domain D_i;
- $C = \{C_1, C_2, ..., C_m\}$ is a set of constraints of the form $c ::= c \vee c \mid \neg c \mid x_i < x_j$ for any $1 \geq i \geq n and 1 \geq j \geq n$ [35][11].

The number of possible configurations without taking constraints into account can be determined by $\prod_{i=1}^{n} |D_i|$. A CSP can be seen as a generalisation of a Boolean Satisfiability Problem (SAT), which is known to be an NP-complete problem [36] [16].

A special type of CSP is interactive configuration, where a user is assisted by a configurator in making choices resulting in a valid configuration. During configuration, a user repeatedly selects an option for a variable until all variables are assigned. Each time a user makes a choice, subsequent choices that cannot result in a valid configuration due to constraints are removed, resulting in the production of only valid configurations. Interactive configuration is [11][12]:

- Backtrack-free: all options that do not lead to a valid configuration should be removed;
- Complete: maximally permissive, all options that lead to a valid configuration should not be removed;
- Real time: should respond immediately.

Questions and answers relate to the variables and domains in the CSP and the constraints to design rules. That is, the set of answers to a question X_i forms the discrete domain D_i. The design rules can be modeled by means of Boolean functions. This way the parameters of the CSP define the solution space. This answers the first research question, next section dives deeper into the implementation of this strategy.

3.2 Configurator calculation

BDD's are a long-standing technique that is widely used to efficiently manage binary logical operations, especially in the area of verification and synthesis [1] [8]. A BDD is a data structure used to represent a Boolean function by a finite Directed Acyclic Graph (DAG) [10].

In a BDD, each node represents an If-Then-Else (ITE) operator, where the condition is a decision variable. This decision variable is Boolean, meaning it can be either true or false. Consequently, each node has two arrows pointing to its child nodes: one arrow is assigned as true, and the other as false. Depending on the combination of decision variables, the outcome for that combination is either true or false.

When a BDD is 'reduced' and 'ordered', it is known as a Reduced Ordered Binary Decision Diagram (ROBDD). Every Boolean function can be represented by a unique ROBDD, also known as the canonical form, which is even more efficient [3].

Figure 3.1 displays two BDD's representations, both for the formula $f(x_1, \ldots, x_6) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee (x_5 \wedge x_6)$. In this figure, a solid line represents the true case, and a dotted line represents the false case. The right BDD is reduced and ordered and is therefore more compact.

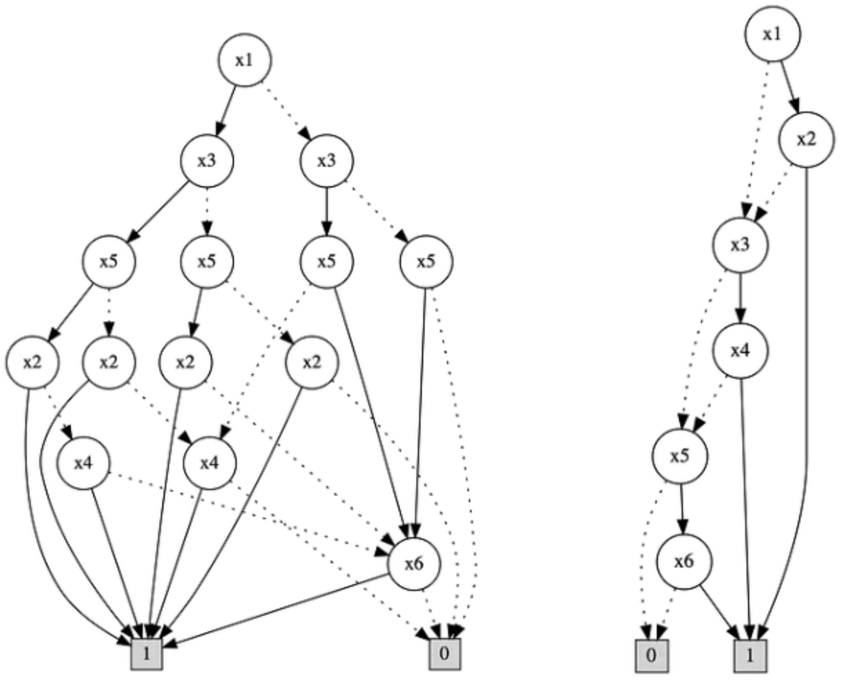


Figure 3.1: Two BDD's representing the same formula [23]

BDD's can be used for constrained programming as a constrained handling technique, usually outperforming other constrained handling techniques [32].

3.3 Supervisory control synthesis

A configurator with only discrete answers for questions can be modelled as a discrete event system. Discrete event systems can be controlled by means of supervisory control theory [31]. Supervisory controllers are often used in systems containing software that controls the operation (of a plant). Often it receives an action from the operator and decides, based on the current state of the plant, whether the action is allowed [9]. The supervisor controllers restrict the system's behavior to align with the requirements, ensuring the execution of events satisfies the given specification [22] [31].

Traditionally, the supervisory controller is manually modeled via a model based engineering approach, which has as downside the manual implementation and verification. Synthesis-based engineering (SBE) is a form of model-based engineering in which a supervisory controller is synthesized (computed algorithmically) [31] [26]. Synthesizing guarantees the requirements are always adhered to, thus verification is not required [26]. Furthermore no manual implementation is required [30].

An analogy can be drawn between a supervisory controller and product configurator. The supervisory controller determines whether a specific action by the engineer is permitted based on specified rules, thereby controlling the requirements specification of a facility (such as a lock). Traditionally, a configurator is also designed manually, however this thesis explores whether the configurator can also be synthesised. By synthesising the configurator, the production of valid configurations is guaranteed, answering the second research question.

Chapter 4

Configurator synthesis

In this chapter, a method is proposed for the automatic generation of a rule based product configurator based on (hierarchical) questions with answers and design rules. An overview of the method is shown in Figure 4.1.

In this method, a generic configurator data structure is defined with which the (hierarchical) questions with answers and design rules can be described. Configurator data structure is then used to synthesize (compute algorithmically) the data for corresponding configurator.This method ensures the production of valid configurations, e.g. configurations that comply with the design rules. Therefore, the design rules can be used to define the solution space of the configurator. Furthermore, the method is suitable for the creation of a configurator for both greenfield and brownfield project.

First, the required configurator data structure is defined. Next, the output rules are defined, followed by an explanation of how the configurator is synthesised. In the last section, it is discussed how the method is used for both greenfield and brownfield projects.

4.1 Configurator data structure

The (hierarchical) questions, answers, and design rules need to be systematically described so that they can be utilized in synthesis. Therefore, a configurator data structure is designed in which (hierarchical) questions with answers and design rules can be described. What has to be described in this configurator data structure and how this is done, is explained in the following paragraphs.

4.1.1 (Hierarchical) Questions and answers

Figure 4.2 shows an example of hierarchical questions. It displays a DAG, with questions QUE-A and QUE-J as distinct root questions. The root questions are the first to be answered, and the arrows indicate the possible answers and subsequent follow-up questions. Question QUE-A has answers ANS-A1 and ANS-A2, leading to follow-up questions QUE-B, QUE-C and QUE-D, QUE-E respectively. Question QUE-J is considered a flat question, since that root question does not have any follow-up questions.

The relations of questions and answers of a DAG (as illustrated in Figure 4.2) need be translated into a configurator data structure to be able to perform the calculations needed for synthesis. The configurator data structure, along with the implementation example from Figure 4.2, is presented in Listing 4.1. The object CF is a nested dictionary. The keys of the outer dictionary represent the questions. The values of the outer dictionary contain the inner dictionaries. The keys of the inner dictionaries represent the answers to the questions in the outer dictionary. The values of the inner dictionaries are lists containing the follow-up questions that need to be answered after the corresponding answer is chosen.

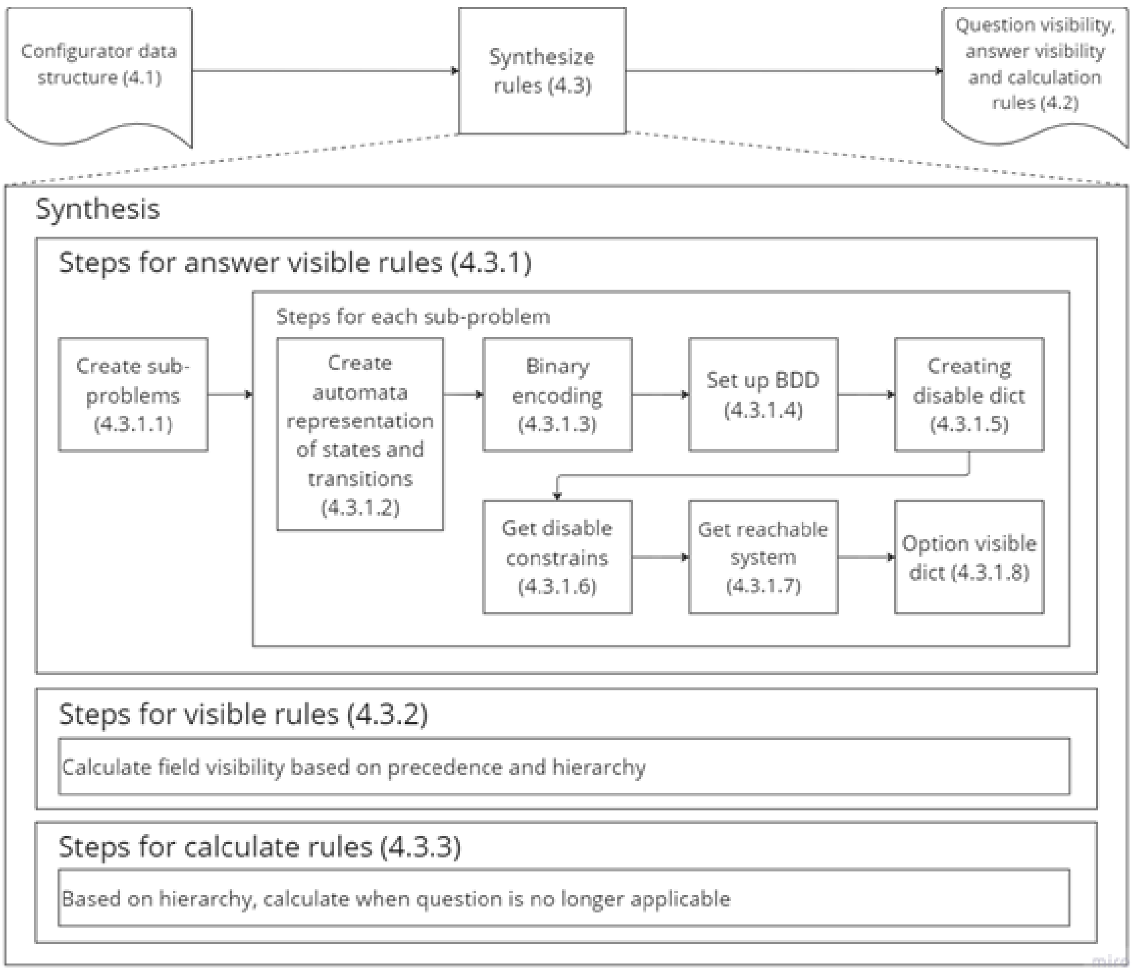

Configurator data structure (4.1)
Synthesize rules (4.3)
Question visibility, answer visibility and calculation rules (4.2)
Synthesis
Steps for answer visible rules (4.3.1)
Steps for each sub-problem
Create sub-problems (4.3.1.1)
Create automata representation of states and transitions (4.3.1.2)
Binary encoding (4.3.1.3)
Set up BDD (4.3.1.4)
Creating disable dict (4.3.1.5)
Get disable constrains (4.3.1.6)
Get reachable system (4.3.1.7)
Option visible dict (4.3.1.8)
Steps for visible rules (4.3.2)
Calculate field visibility based on precedence and hierarchy
Steps for calculate rules (4.3.3)
Based on hierarchy, calculate when question is no longer applicable

Figure 4.1: Overview of the method

Hierarchical and non-hierarchical questions with answers are described using this configurator data structure and thus it is suitable to describe the product platform from previous chapter. The hierarchical order of the questions with answers imply design rules. The hierarchy is rewritten in terms of design rules and non-hierarchical questions. How this is done will be further explained in upcoming sections.

4.1.2 Additional design rules

Hierarchical questions already imply that some combinations can not be made and imply some sequence in which questions must be answered. However, the hierarchy might not be sufficient to restrict all invalid configurations or to prescribe the sequence in which the questions must be answered.

The configurator data structure should also be able to describe combinations of answers that may not be selected together, must be selected together and prescribe a sequence in which the questions must be answered. To this aim, three additional design rules are introduced:

- Excludes: combination of two or more answers that may not be selected together;
- Enforces: combination of two or more answers that must be selected together;
- Precedence: question that must be answered before another question may be answered.

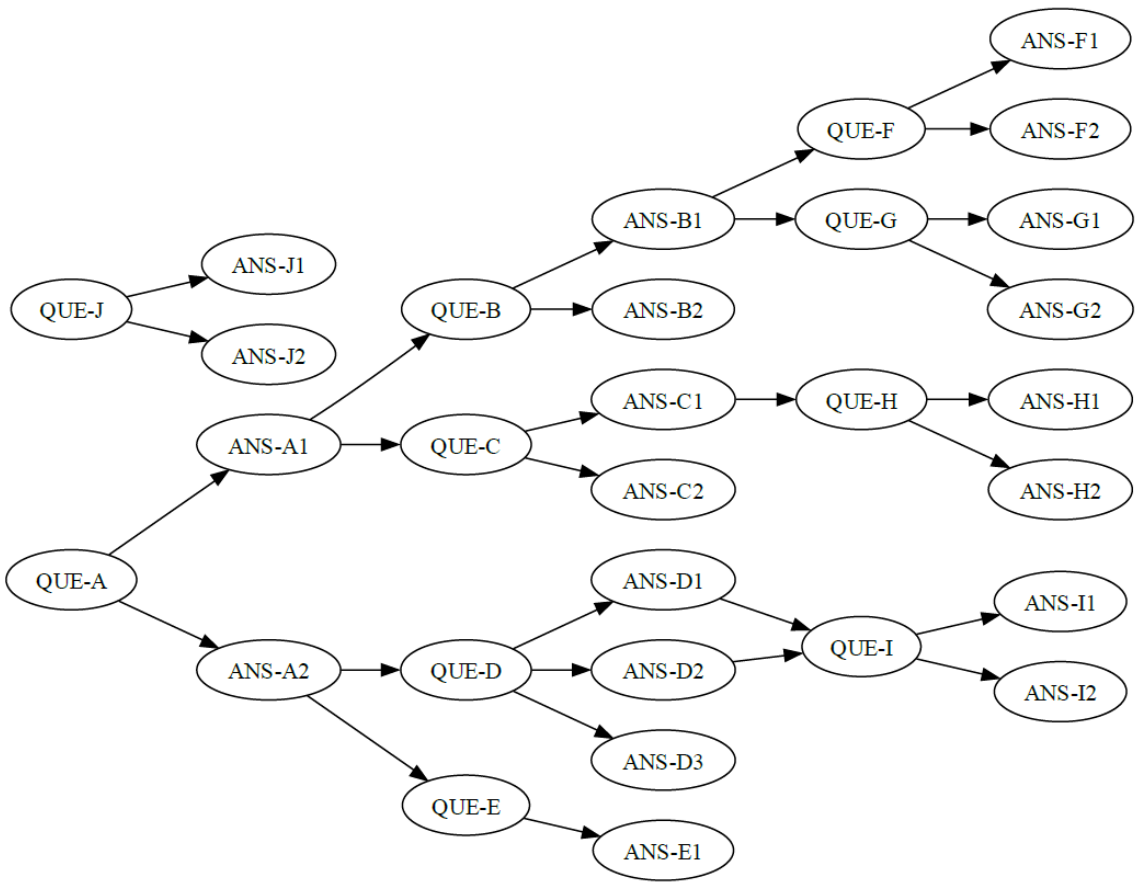


Figure 4.2: Example DAG

The configurator data structure for the excludes and enforces are displayed in Listings 4.2 and 4.4 respectively. The configurator data structure for excludes is equal to that of the enforces. It is a list of lists with inside the inner list, tuples containing question and answer pairs of the excluded/enforced combinations. Example excludes and enforces are displayed in Listings 4.3 and 4.5. In those examples combination F.F1 and G.G1 and combination B.B2, H.H1 and J.J1 are excluded and D.D1 and I.I1 are enforced respectively.

The data structure used for precedence is displayed in Listing 4.6. It is a dictionary containing questions as keys and a list of questions as value. First, the key must be answered before any of the questions in the value may be answered. An example precedence is shown in Listing 4.7. In this example, question J must be answered before an answer may be given to question A. Furthermore, question F must be answered before an answer may be given to question G.

4.2 Question visibility, answer visibility and calculation rules

The output of the synthesis are the questions with answers and configurator rules that describe how the configurator behaves. These configurator rules are written in the form IF *condition* THEN *consequence*, and is directly implemented in a rule based configurator. Three types of configurator rules are used, namely answer-visible rules, question-visible rules and calculate rules.

Answer-visible rules are used to determine, based on current selections in the configurator, whether an answer may be selected or not. An answer may only be selected if the selection does

Listing 4.1: Example configurator data structure for questions and answers

```
cf: Dict[str, Dict[str, List[str]]]

cf = {
    "A": {"A1": ["B", "C"], "A2": ["D", "E"]},
    "B": {"B1": ["F", "G"], "B2": []},
    "C": {"C1": ["H"], "C2": []},
    "D": {"D1": ["I"], "D2": ["I"], "D3": []},
    "E": {"E1": []},
    "F": {"F1": [], "F2": []},
    "G": {"G1": [], "G2": []},
    "H": {"H1": [], "H2": []},
    "I": {"I1": [], "I2": []},
    "J": {"J1": [], "J2": []}
}
```

Listing 4.2: Configurator data structure for excludes

```
excludes: List[List[Tuple[str]]]
```

Listing 4.3: Example configurator data structure for excludes

```
excludes = [
    [("F", "F1"), ("G", "G1")],
    [("B", "B2"), ("H", "H1"), ("J", "J1")]
]
```

Listing 4.4: Configurator data structure for enforces

```
enforces: List[List[Tuple[str]]]
```

Listing 4.5: Example configurator data structure for enforces

```
enforces = [
    [("D", "D1"), ("I", "I1")]
]
```

Listing 4.6: Configurator data structure for precedence

```
precedence: Dict[str, List[str]]
```

Listing 4.7: Example configurator data structure for precedence

```
precedence = {
    "J": ["A"],
    "F": ["G"]
}
```

not result in a configuration with restricted answer combinations. If the answer-visible rules are synthesised correctly, they guarantee the production of valid configurations. Since answer-visible rules only restrict restricted answer combinations, the configurator is maximal permissive.

The sequence in which the questions are answered, does not influence the production of valid configurations. Therefore, answer-visible rules do not restrict the user from answering questions in the wrong sequence. However, it is desired to guide the user through the configurator. To this aim, question-visible rules are synthesised. Question visible-rules selectively show or hide questions to help the user walk through the configurator.

Furthermore also calculate rules are synthesised. Calculate rules calculate "Not applicable" for questions that become irrelevant (and thus invisible) due to the hierarchy of questions. This has two advantages: first it makes it easy to check if all questions have been answered, and second the "Not applicable" answer can be used in design rules.

Three data structures are introduced to generically describe configurator rules:

- Option visible dictionary: contains for each answer the set of states when that answers is visible;
- Option field visible dictionary: contains information for each field when it is visible;
- Option calculate NA dictionary: contains information when "Not applicable" (referred to as NA) should be calculated.

The questions with answers and configurator rules are later used to generate a configurator. Hence this will be used to generate a Merkato template from.

4.3 Synthesis of the rules

This section describes how, based on the configurator data structure, the method synthesises the configurator rules.

4.3.1 Answer-visible rules

In this section the steps for the synthesis of answers-visible rules are explained.

4.3.1.1 Create subproblems

Due to the binary nature of BDD's, design rules are expressed in Boolean functions. Answer-visible rules are also expressed in Boolean functions, because for a specific Boolean condition the specific answer is visible (true) or invisible (false). Furthermore, BDD's offer several advantages as described in Chapter 3 and are therefore used for synthesis of the option-visible rules based on the design rules.

Due to the fact that BDD calculation time grows exponentially, creating a single BDD for this problem takes to long to solve. Instead of creating a single BDD, a BDD is made for each question, referred to as the target question. To the BDD for a target question, a subset of the questions, answers, exclude and enforce rules that influence the question's answer-visible rules are added. These relatively small BDD's still give a valid answer, however the calculation time is significantly reduced due to the fact that solving multiple simple BDD's scale linearly in time.

Listing 4.8: Example derived excludes from Figure 4.2

```
Excludes_derived_from_hierarchy = [
    [('A', 'A2'), ('C', 'C1')], [('A', 'A2'), ('C', 'C2')],
    [('A', 'A1'), ('D', 'D1')], [('A', 'A1'), ('D', 'D2')],
    [('A', 'A1'), ('D', 'D3')], [('A', 'A1'), ('E', 'E1')],
    [('A', 'A2'), ('B', 'B1')], [('A', 'A2'), ('B', 'B2')],
    [('B', 'B2'), ('G', 'G1')], [('B', 'B2'), ('G', 'G2')],
    [('B', 'B2'), ('F', 'F1')], [('B', 'B2'), ('F', 'F2')],
    [('C', 'C2'), ('H', 'H1')], [('C', 'C2'), ('H', 'H2')],
    [('D', 'D3'), ('I', 'I1')], [('D', 'D3'), ('I', 'I2')],

    [('A', 'A1'), ('B', 'NA')], [('A', 'A1'), ('C', 'NA')],
    [('A', 'A2'), ('D', 'NA')], [('A', 'A2'), ('E', 'NA')],
    [('B', 'B1'), ('F', 'NA')], [('B', 'B1'), ('G', 'NA')],
    [('C', 'C1'), ('H', 'NA')], [('D', 'D1'), ('I', 'NA')],
    [('D', 'D2'), ('I', 'NA')]
]
```

Initially, no answer has been selected for questions. However, this empty state can not be used in Boolean functions. Therefore the answer "undefined" is added to all questions and set as initial answer. Furthermore, to ensure the validity of the configuration, a configuration is finished if every question has been answered. Therefore, none of the questions are undefined.

Hierarchical questions are parsed into flat questions and a set of design rules. As a result, only flat questions and design rules are remaining and are easier to use in the Boolean functions. The design rules derived are exclude- and precedence rules for each question and its direct follow-up questions. Answer-visible rules do not restrict the user from answering questions in the wrong sequence and therefore, for the synthesis of the answer-visible rules, only exclude rules are considered. Furthermore, branches for not chosen questions do not have to be answered. The questions in that branch become not applicable. For the questions that can become not applicable, the answer "Not applicable" is added. This answer should be automatically selected or deselected when a question becomes non applicable or applicable respectively. This automatic selection is done via calculate rules as further explained in Section 4.3.3. The answer "Not applicable" distinguishes it from "Undefined", such that it is known when a configuration is finished and it can be used in design rules. For the "Not applicable" answers, also exclude rules can be derived from the hierarchy. To illustrate this, the exclude rules are derived from Figure 4.2 and shown in Listing 4.8. Question QUE-C is the follow-up question of answer ANS-A1, therefore all answers for question QUE-C are excluded for all answers of question QUE-A other then ANS-A1. Furthermore since question QUE-C is a follow-up question of answer-A1, QUE-C may not be "Not applicable". In this listing, excludes using "Not applicable" are shown after the break line. BDD's are used to ensure the exclusiveness of the option-visible rules. Meaning that each answer only has unique conditions under which the answer is visible.

To a BDD for a specific question, only exclude and enforce rules are added in which the specific question is mentioned together with the questions. An exception to this are the exclude rules derived from hierarchy between the question and its follow-up questions, because questions already take parent questions into account. A question is always added together with its options.

This is illustrated for question D in Listing 4.9. This subproblem only contains three questions, namely the target question and the other questions mentioned in the subproblems excludes and enforces. The questions contain all corresponding answers (except for undefined, because that is added in the next step). Follow-up questions are removed since constraints of the parsed hierarchy are added to the exclude list. Looking at the exclude and enforce rules in Listings 4.3 and 4.5, it can be seen that target question D is only mentioned once in the enforce rule. Therefore, only this

Listing 4.9: Example relevant subproblem for D

```
CF = {
    'A': {'A1': [], 'A2': []},
    'I': {'I1': [], 'I2': [], 'NA': []},
    'D': {'D1': [], 'D2': [], 'D3': [], 'NA': []}
    }

Excludes = [
    [('A', 'A1'), ('D', 'D1')], [('A', 'A1'), ('D', 'D2')],
    [('A', 'A1'), ('D', 'D3')], [('A', 'A2'), ('D', 'NA')]
]

Enforces = [
    [('D', 'D1'), ('I', 'I1')]
]
```

Listing 4.10: Example automata for subproblem D

```
{
    'D': {'S': ['D1', 'D2', 'D3', 'NA', 'UN'],
        'E': ['D1', 'D2', 'D3', 'NA', 'UN']},
    'I': {'S': ['I1', 'I2', 'NA', 'UN'],
        'E': ['I1', 'I2', 'NA', 'UN'],
    'A': {'S': ['A1', 'A2', 'UN'],
        'E': ['A1', 'A2', 'UN']}
}
```

enforce is added to the subproblem. The steps in following sections are done for each sub BDD.

4.3.1.2 Create automata representation of states and transitions

In a BDD, only binary variables and Boolean constraints can be added. Questions with answers are modelled using variables representing the answers together with the constraints that a single answer for each question is selected at all times. Each variable is symbolized by a single answer, with a variable set to 1 indicating the selection of that answer. Those variables are referred to as state variables and are used to describe the current state of the sub-problem.

Based on the state variables, it is determined which other answers for the target question may be selected. The selection of an answer is referred to as an event. However, to indicate if the event of selecting another answer is possible, another variable for each answer is required. If that event variable is set to 1, the event of selecting the answer represented by that variable can take place.

This automata representation is illustrated for the subproblem of target question D in Listing 4.10. It can be that each question constraints a list for the states and events indicated by 'S' and 'E' respectively.

4.3.1.3 Binary encoding of subproblem

In this section a slightly more advanced way of representing the state of a question is introduced, which can reduce the amount of variables the subproblem significantly.

Since only one of the answers to a question is selected at all times, only describing the selected answer is sufficient. The binary encoding of the answers is done according to [22], reducing the

Listing 4.11: Example binary encoding for subproblem D

```
{
    'A': {
        'S': {'A1': [0, 0], 'A2': [0, 1], 'UN': [1, 0]},
        'E': {'A1': 2, 'A2': 3, 'UN': 4}
    },
    'D': {
        'S': {'D1': [0, 0, 0], 'D2': [0, 0, 1], 'D3': [0, 1, 0],
            'NA': [0, 1, 1], 'UN': [1, 0, 0]},
        'E': {'D1': 8, 'D2': 9, 'D3': 10, 'NA': 11, 'UN': 12}
    },
    'I': {
        'S': {'I1': [0, 0], 'I2': [0, 1], 'NA': [1, 0], 'UN': [1, 1]},
        'E': {'I1': 15, 'I2': 16, 'NA': 17, 'UN': 18,}
    }
}
```

amount of variables per question according to Equation 4.1. This reduction repeats itself for each question in the subproblem and therefore reduces the amount of variables significantly. Especially when a subproblem contains question(s) with many answers.

$$amount\_of\_vars\_after\_encoding = \lceil \log_2(amount\_of\_vars\_before - encoding) \rceil \tag{4.1}$$

It might be that not all combinations of bits are assigned to a state and thus these unassigned bit combinations are blocked. This is achieved by adding a constrained for each unassigned bit combination, stating that bit combination may not occur.

The events are not encoded in binary, because there may be multiple events possible from a particular state. Therefore, each event is encoded by a single Boolean variable.

The binary encoding is illustrated for the subproblem of target question D in Listing 4.11. It can be seen that every state is binary encoded. Furthermore, each event is assigned a number that refers to its position among the variables. The states of A are binary encoded by variables [X0, X1]. Since the states of A require the first two variables, the events for A start counting at 2. In total, for question D, 19 variables are utilized in the BDD. The allowed events for a specific state are represented by 19 bit. Furthermore, it can be seen that to state [X0=1,X1=1] no answer has been assigned. Therefore, variable X0 and variable X1 may not be set to 1 both at the same time. Hence the constraint 'not (X0 and X1)' is added.

4.3.1.4 Set up BDD

In setting up a BDD, the library [6] is used. This library constructs BDD's based on provided variables and Boolean functions and takes care of the mathematics involved such as managing the ordering and reduction. Additionally, the library facilitates BDD evaluation, returning for example all possible combinations of variables for which the expression evaluates in true.

4.3.1.5 Creating disable dictionary

Exclude and enforce rules only contain the states that may not or must be selected together respectively. To prevent invalid combinations to occur together, the event resulting in a invalid combination is restricted. The conditions for which an event should be restricted can be derived from exclude and enforce constraints.

Listing 4.12: Example disabled dictionary for excludes

```
disable_dict = {
    ('F', 'F1'): [[('G', 'G1')]],
    ('G', 'G1'): [[('F', 'F1')]],

    ('B', 'B2'): [[('H', 'H1'), ('J', 'J1')]],
    ('H', 'H1'): [[('B', 'B2'), ('J', 'J1')]],
    ('J', 'J1'): [[('B', 'B2'), ('H', 'H1')]]
}
```

Listing 4.13: Example disabled dictionary for enforces

```
disable_dictionary = {
    ('D', 'D1'): [[('I', 'I2')], [('I', 'NA')]],
    ('D', 'D2'): [[('I', 'I1')]],
    ('D', 'D3'): [[('I', 'I1')]],
    ('D', 'NA'): [[('I', 'I1')]],
    ('I', 'I1'): [[('D', 'D2')], [('D', 'D3')], [('D', 'NA')]],
    ('I', 'I2'): [[('D', 'D1')]],
    ('I', 'NA'): [[('D', 'D1')]]
}
```

A dictionary is created with as key the event and as value a list with states in which the event is restricted. The restricted states are easily derived from exclude rules, since for each exclude, each state is set as event taking the other states in that exclude as condition in which the event should be restricted. The derivation from enforce rules is a bit more difficult, since an enforce rule implies multiple exclude rules. Enforcing a combination is the same as excluding all other answer pairs for the questions mentioned.

This disabled dictionary is only made for each subproblem and will therefore only contain the exclude and enforce rules for that subproblem. For illustrating purposes the disabled dictionary is derived for the exclude rules from Listing 4.3 in Listing 4.12. The break line separates the two exclude rules. It can be seen that for each state in an exclude, the corresponding event is used as key and other states as condition. Furthermore, the disabled dictionary is also derived from Listing 4.5 in Listing 4.13, which uses an enforce rule instead of an exclude rule. A combination is enforced if everything else is excluded. From this disabled dictionary, it can be seen that the only combination of answers for questions QUE-D and QEE-I, that can be selected together, is the enforced combination from Listing 4.5.

The disabled dictionary that is derived for subproblem D (Listing 4.9) is shown in Listing 4.14. Each event that has a blocking condition is mentioned in this listing, which is in this case all events. Note that the disabled dictionary in Listing 4.13 is part of 4.14, since question QUE-D is mentioned in the enforce in Listing 4.5.

4.3.1.6 Get disable constraints

An event in the disabled dictionary is restricted when one or more of the corresponding conditions is currently selected. If none of conditions is currently selected, the event is allowed (making it maximally permissive). A Boolean function is made used to determine if an event should be restricted or not. This Boolean function is illustrated by the logic gate diagram in Figure 4.3.

Listing 4.14: Disabled dictionary for subproblem D

```
disable_dictionary = {
    ('A', 'A1'): [[('D', 'D1')], [('D', 'D2')], [('D', 'D3')]],
    ('D', 'D1'): [[('A', 'A1')], [('I', 'I2')], [('I', 'NA')]],
    ('D', 'D2'): [[('A', 'A1')], [('I', 'I1')]],
    ('D', 'D3'): [[('A', 'A1')], [('I', 'I1')]],
    ('A', 'A2'): [[('D', 'NA')]],
    ('D', 'NA'): [[('A', 'A2')], [('I', 'I1')]],
    ('I', 'I1'): [[('D', 'D2')], [('D', 'D3')], [('D', 'NA')]],
    ('I', 'I2'): [[('D', 'D1')]],
    ('I', 'NA'): [[('D', 'D1')]]
}
```

To check whether one or more of the corresponding conditions for an event is currently selected, an 'OR'-function is used between the conditions. Each condition can consist of multiple states which are modelled using an 'AND'-function. Furthermore, another 'AND' function is used for the binary encoding, because a single answer is described by one or more bit. To ensure either the event is true/possible or the one of the blocking conditions is true, an 'XOR' function is used. Also restrictions are added to make sure the not-used Boolean variables can not be selected. Unconstrained answers are always set to be true, thus always visible.

All constraints are applied at the same time. The 'XOR' for all events, together with the restrictions for the not used Boolean variables and unconstrained events must together be true for a allowed configurator state. This is done using another 'AND'-function.

4.3.1.7 Get reachable system

Not all conditions for visible answers, derived from the BDD model, may occur because the event leading to that condition is blocked. It is preferable to derive only relevant conditions so that the outcome is more concise and faster.

To determine which states are in the reachable system, a breadth-first search is conducted, starting from the initial state where all questions are undefined. Based on the BDD model, it is determined which events can occur from the current state. From each state found, a check is made again to see which events are allowed from that state. The found states are stored.

When all reachable states have been found, it is added to the BDD that only states from the reachable states are allowed to occur together. This ensures that only the answer visible conditions that occur in the reachable system remain.

The reduction achieved by only considering the reachable system for the problem described in Listing 4.1 decreases the number of answer-visible rules from 1857 to 1522 (-18%). Given that the state space grows exponentially, this is a significant reduction.

4.3.1.8 Option visible dictionary for subquestion

From the BDD, the answer-visible dictionary is computed. This is done for each event for the target question. For each evaluation of the option-visible conditions, another 'and' constrain is added between the reachable system BDD and the current event itself. A method from the BDD library is applied to obtain all assignments for variables of which the BDD evaluates in true. The assignments for variables for a specific event are the conditions for which the event is possible.

Listing 4.15: Example option for A and E

```
('A', defaultdict(<class 'set'>, {
'A1': {(('A', 'A2'),), (('A', 'A1'),), (('A', 'UN'),)},
'A2': {(('A', 'A2'),), (('A', 'A1'),), (('A', 'UN'),)},
'UN': {(('A', 'A2'),), (('A', 'A1'),), (('A', 'UN'),)}
}))


('E', defaultdict(<class 'set'>, {
'E1': {(('E', 'UN'), ('A', 'A2')), (('E', 'E1'), ('A', 'UN')),
(('E', 'UN'), ('A', 'UN')), (('E', 'NA'), ('A', 'UN')),
(('E', 'E1'), ('A', 'A2'))},
'NA': {(('E', 'E1'), ('A', 'UN')), (('E', 'UN'), ('A', 'UN')),
(('E', 'UN'), ('A', 'A1')), (('E', 'NA'), ('A', 'UN')),
(('E', 'NA'), ('A', 'A1'))},
'UN': {(('E', 'UN'), ('A', 'A2')), (('E', 'E1'), ('A', 'UN')),
(('E', 'UN'), ('A', 'UN')), (('E', 'UN'), ('A', 'A1')),
(('E', 'NA'), ('A', 'UN')), (('E', 'E1'), ('A', 'A2')),
(('E', 'NA'), ('A', 'A1'))}}))
```

In Listing 4.15 an example is given of how the answer-visible dictionary for target questions A and E respectively. The option mentioned as key is visible if one of the conditions, mentioned as value, is true.

4.3.2 Question-visible rules

Questions are shown in a specific order to guide the user through the configuration process. Question-visible rules determine the visibility of questions and are derived from hierarchy and additional precedence rules. The question visibility rules are calculated separately from the answers. This reduces the amount of variables in the BDD's, while the answer-visibility rules still guarantee the production of valid configurations.

A question is visible if it is a follow-up question to the answer chosen for the parent question and if the answer 'Not Applicable' is not chosen (does not apply to root questions). Furthermore all questions mentioned as a key in the precedence must be answered before the questions in corresponding value are visible. The information for question-visible rules is saved in a question visible dictionary. This dictionary contains questions as keys and the conditions in which the questions are visible as corresponding values. An example of the question-visible dictionary for Listing 4.1 is given in Listing 4.16.

4.3.3 Calculate rules

To some questions the answer "Not applicable" is added. These questions might be not applicable because it is an follow-up question. If a question can only be answered with "Not applicable", it is preferred to automatically calculate "Not applicable", without asking the question. Therefore, in previous section it is explained that these questions are not visible. Otherwise, users would have to answers much more questions. Furthermore, if the user changes the answer of the parent question to an answer that requires a follow-up question, the automatic calculation is undone. However, since the "Not applicable" answer is not visible if an answer requires a follow-up question, this should happen automatically. This is illustrated in Figure 4.2. If answer A1 is chosen, then answers to questions B and C must be selected, which may not be "Not applicable". Furthermore, since A2 has not been chosen, no answer for questions D and E may be chosen and should be automatically set to "Not applicable".

Listing 4.16: Example visible dictionary

```
defaultdict(<class 'set'>, {
    'A': {('B', 'NA'), ('C', 'NA'), ('D', 'NA'), ('E', '==""'),
        ('E', 'NA'), ('C', '==""'), ('J', '!=""'),
        ('B', '==""'), ('D', '==""')},
    'B': {('F', 'NA'), ('G', '==""'), ('F', '==""'), ('G', 'NA'),
        ('A', 'A1')},
    'F': {('G', '==""'), ('G', 'NA'), ('B', 'B1')},
    'G': {('F', '!=""'), ('B', 'B1')},
    'C': {('H', '==""'), ('A', 'A1'), ('H', 'NA')},
    'H': {('C', 'C1')},
    'D': {('I', '==""'), ('I', 'NA'), ('A', 'A2')},
    'I': {('D', 'D2'), ('D', 'D1')},
    'E': {('A', 'A2')},
    'J': {('A', '==""')}
})
```

Listing 4.17: Example calculate dictionary

```
calculate_dictionary = {
    'C': {('A', 'A2')},
    'D': {('A', 'A1')},
    'E': {('A', 'A1')},
    'B': {('A', 'A2')},
    'F': {('B', 'B2'), ('B', 'NA')},
    'G': {('B', 'B2'), ('B', 'NA')},
    'H': {('C', 'C2'), ('C', 'NA')},
    'I': {('D', 'NA'), ('D', 'D3')}
}
```

For the automatic selection of "Not applicable" a calculate dictionary is made as shown in 4.17. This dictionary contains questions as keys and the corresponding values are the conditions for which the option "Not applicable" should be automatically selected.

4.4 Configuration process for greenfield and brownfield

As explained in the introduction, RWS wants to use the configurator for greenfield and brownfield projects. In the above steps it is explained how based on the configurator data structure, a configurator is synthesised. However until now it is only mentioned the configurator data structure is used for product platform, LSR and design rules. This section explained how the configurator data structure is utilised for greenfield and brownfield.

The proposed process is displayed in Figure 4.4. For each configuration, the LSR and the PT are required. In the PT indicates whether it is a greenfield or brownfield project. The question "Project type" is added with answers "Greenfield" and "Brownfield". If it is a greenfield project, the future situation can be configured directly, which dependents on the LSR. If it is a brownfield project, the current situation and the component scope must be chosen first, before selecting the future situation, which then depends on the LSR, current situation and component scope. The questions and answers for the current situation and component scope are always in the configurator, but are only visible for brownfield projects.

The current and future situations are specified by choosing variants for the components from the product platform. Therefore the product platform is added twice to the configurator data structure, with the distinguishing prefix "CUR" and "FUT" for the current- and future situation respectively. The hierarchy for both the current- and future situation from the product platform is retained. For the component scope, components of the product platform are added to the configurator data structure with the prefix "SCOPE-CUR", together with the answers "Retain", "Renovate" and "Replace". For optional components, the additional answers are "Add" and "Remove".

At this point, the questions and answers for the PT, LSR, current situation, component scope, and future situation are in the configurator data structure. To ensure it behaves as intended for greenfield and brownfield projects, the following constraints are necessary:

- Follow-up questions for PT greenfield are the "ENV" and "FUT" root questions and for brownfield the "ENV", "CUR", "FUT" root questions;
- Each question in the current situation has a corresponding scope question. This corresponding scope question is set as follow-up question to all answers of the question in the current situation (this way, it does not show for "Not applicable");
- Precedence constraints stating the scope of a component must be chosen before corresponding future situation is chosen;
- Variants for components with a scope of "Retain" or "Renovate" may not change. Therefore, exclude constraints are added. For "Replace" no additional constraints are added. Furthermore for optional components constraints are added such that: if "None" is selected, the scope cannot be "Renovate," "Retain," "Replace," or "Remove," but only "Add" or "None". If "None" is not selected, "None" or "Add" cannot be chosen, but "Renovate", "Retain", "Replace", or "Remove" can be chosen.

An example of the configurator data structure for a greenfield and brownfield configurator for the example product platform and design rules in this chapter is given in Appendix B. A script is developed that automatically derived and adds these constraints. In the next chapter, the application of this method is illustrated on a lock head.

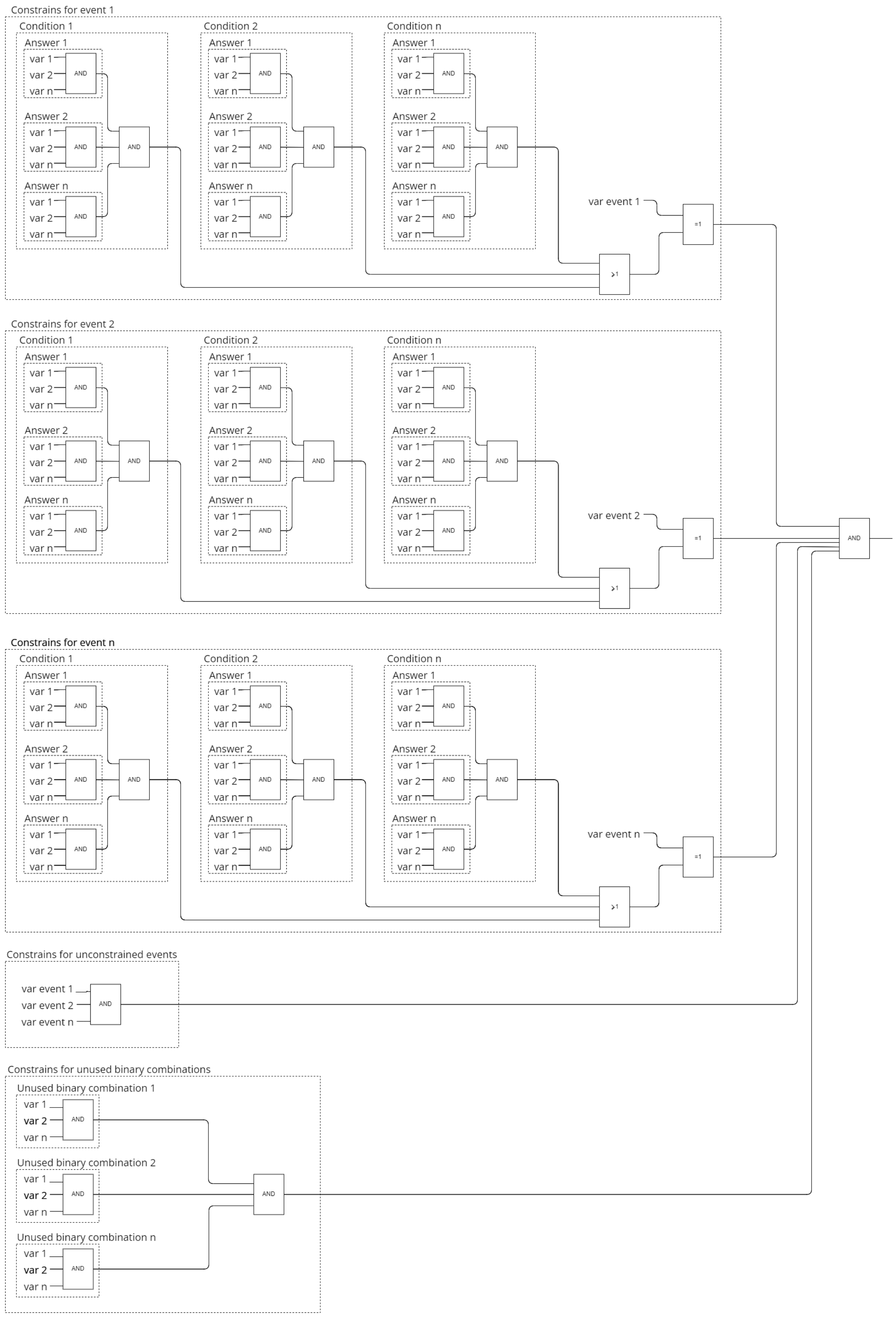


Figure 4.3: Boolean logic of constraints

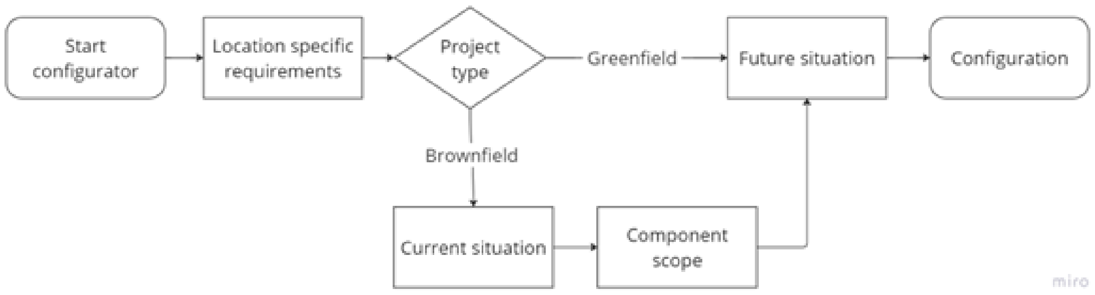


Figure 4.4: Flowchart configuration process

Chapter 5

Application of the method

This chapter demonstrates the method that is proposed in this thesis by means of an example within the context of RWS. It illustrates how an configurator data structure is set up for a lock head configurator that supports greenfield and brownfield and how it is used to synthesize configurator rules. Furthermore it is shown how questions, answers and configurator rules can be implemented in the product configuration software Merkato, which is currently used to facilitate the production of valid configurations. Finally, it is shown how these configurations can result in requirements specifications. Examples are provided, showcasing the method's application in developing a configurator for the most complex component of a lock: the lock head, as shown in Figure 2.3.

5.1 Generation of configurator data structure

A configurator data structure is made for a lock head configurator, which supports the configuration of greenfield and brownfield lock heads. The configurator data structure can be derived from the product platform, LSR and design rules as illustrated in Figure 1.2. The product platform is exported from GRIP, LSR are listed in Section 2.3 and design rules are derived from handbooks and experts. The generation of the configurator data structure is automated in a Python script.

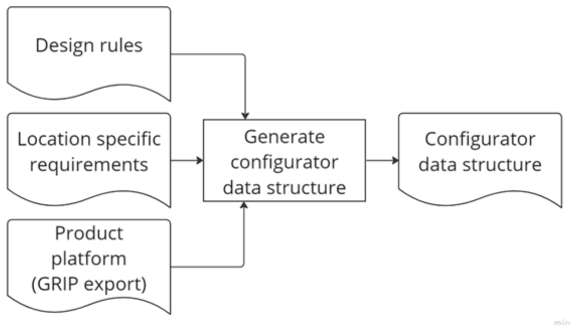

Design rules
Location specific requirements
Product platform (GRIP export)
Generate configurator data structure
Configurator data structure

Figure 5.1: Generation of configurator data structure

For the creation of the configuration data structure of a lock head configurator, the product platform from Figure 2.3 and the LSR from table 5.1 are used. Furthermore, to make this configurator data structure suitable for greenfield and brownfield, the steps from section 4.4 are followed. Additionally, design rules are made for the following requirements:

- Variants "Miter gates" (Puntdeuren) and "Double miter gates" (Vierkantkerende puntdeuren) may only be selected with variants "Electrohydraulic cylinder";(Electrohydraulische cilinder) or "Electromechanic cylinder" (Elektromechanische cilinder).

Table 5.1: Location specific requirements for the lock head

Questions	Answers
Lock head width	≤ 20 m
	> 20 m
Water retention	Mono-directional
	Bi-directional
Water level difference	≤ 4 m
	> 4 and ≤ 6 m
	> 6 m

- Variant "Liftgate" (Hefdeur) may only be selected with variant "Vertical winch installation" (Vertikaal lierwerk);
- Variant "Rollinggate" (Roldeur) may only be selected with variant "Horizontal winch installation" (Horizontaal lierwerk);
- Variant "Rinket" is used for a "Water level difference ≤ 4 m", unless the current variant is "Culvert" (Omloopriolen) of which the component scope is "Retain" or "Renovate";
- Variant "Culvert" is used for a "Water level difference > 6 m", unless the current variant is "Rinkets" (Rinketten) of which the component scope is "Retain" or "Renovate";
- For a "Lock head width ≤ 20 m" only variants "Miter gates" or "Double miter gates" may be used;
- For a "Lock head width > 20 m" only variants "Lift gates" or "Rolling gates" may be used;
- For a "Water retention == Mono-directional", variant "Double miter gates" may not be used;
- For a "Water retention == Bi-directional", variant "Single miter gates" may not be used.

This information is put in the configurator data structure.

5.2 Synthesis of the rules

The configurator data structure is used to synthesise the configurator rules. The synthesised lock head configurator contains a total number of 25 questions, 69 answers and 22136 answer-visible rules. This is calculated under 10 seconds using Python on a commercial laptop from 2021.

Furthermore the method has also been tested on the complete lock product platform, however to this, many design rules are missing. The synthesised lock configurator contains a total number of 597 questions, 1940 answers and 285718 answer-visible rules. This is calculated in approximately 1 minute using Python on a commercial laptop from 2021.

The time it takes to generate a configurator is unmatched by manual development of a configurator, which often takes weeks or months. Moreover, this method guarantees that the generated configurator rules are valid by using the BDD's, unlike the conventional approach.

5.3 Generation of Merkato template

In this demonstration, the configurator data is stored into the Merkato configuration software, which is used for the production of configurations. A configurator can be imported and exported using templates with a specific XML format. XML (eXtensible Markup Language) is a flexible text-based format used for structuring, storing, and transporting data[25]. This XML-file is generated based on the questions, answers and configurator rules in Python. This process is

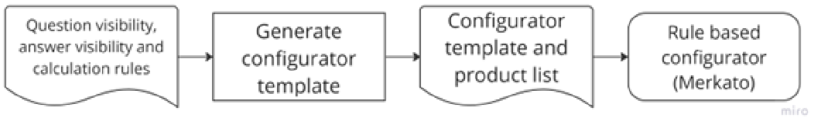

Question visibility, answer visibility and calculation rules
Generate configurator template
Configurator template and product list
Rule based configurator (Merkato)
miro

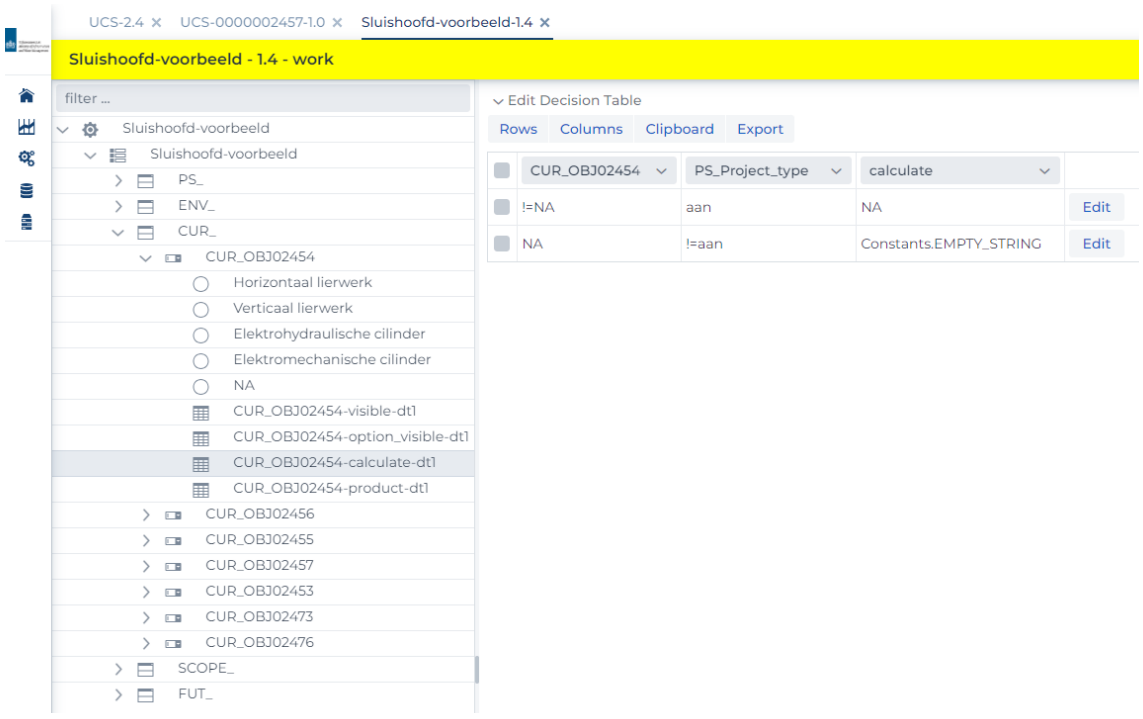
Figure 5.2: Generate configurator data structure

visualised in Figure 5.2.

Flat questions and answers are used to create XML information for fields and options in Merkato. Furthermore, questions are grouped in the XML based on the given prefixes, such that they appear in a separate section in Merkato. Currently all sections are placed on the same page.

To the questions, the following decision tables are added: Calculate tables, Visible tables, Option-visible tables and Product tables. The first three tables contain the same information as the created questions-visibility-, answer-visibility- and calculate rules as calculated in previous the step. Merkato uses a slightly different notation for the rules, therefore the rules are converted. Implemented examples for these tables can be seen in Figures 5.3, 5.4 and 5.5.

Merkato determines, based on the chosen answers and product tables, which products are necessary and puts those products on the quote list. For the current situation, component scope and future situation, product tables are generated to put the right products on the quote list. The implemented product table can be seen in Figure 5.6. For questions about the current- and future situation, this table puts the variants that correspond to the chosen answers as item on the quote list. For component scope questions, the item corresponding to the combination of current situation and chosen scope is put on the quote list. The necessary products are created in previous section. The goal of the products on the quote list is explained in upcoming sections.

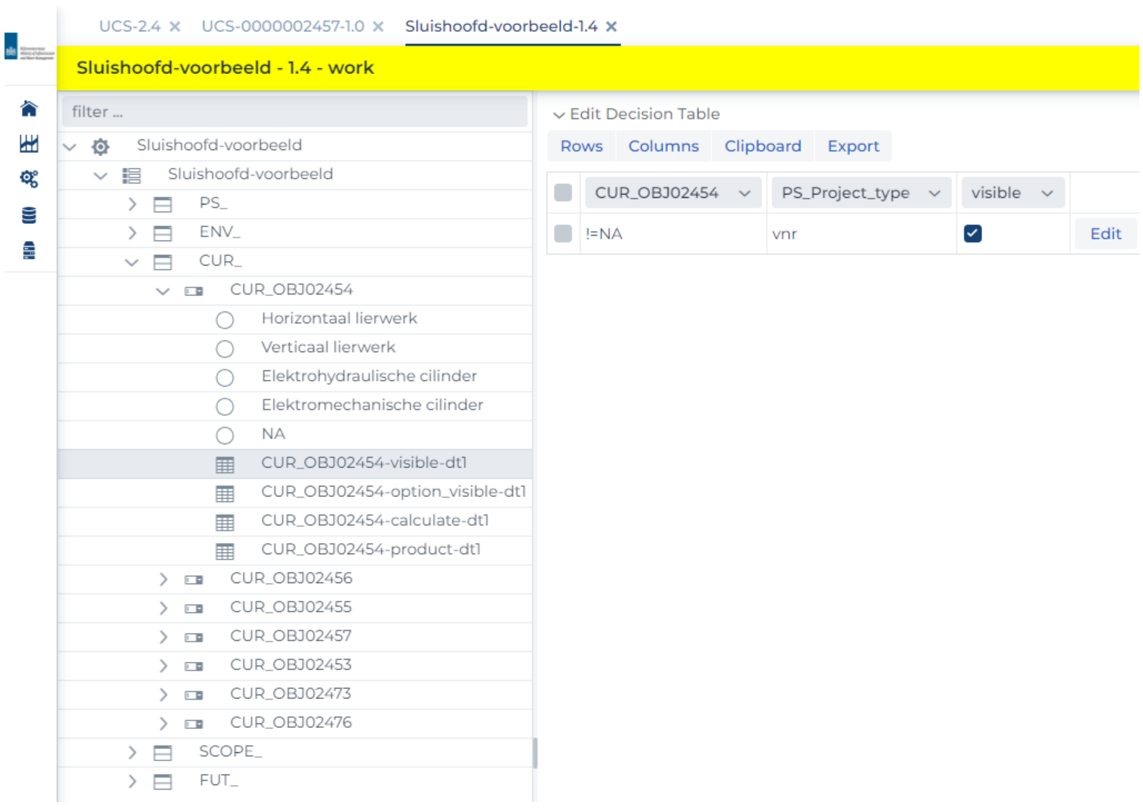


Sluishoofd-voorbeeld - 1.4 - work

- Sluishoofd-voorbeeld
  - Sluishoofd-voorbeeld
    - PS_
    - ENV_
    - CUR_
      - CUR_OBJ02454
        - Horizontaal lierwerk
        - Verticaal lierwerk
        - Elektrohydraulische cilinder
        - Elektromechanische cilinder
        - NA
        - CUR_OBJ02454-visible-dt1
        - CUR_OBJ02454-option_visible-dt1
        - CUR_OBJ02454-calculate-dt1
        - CUR_OBJ02454-product-dt1
      - CUR_OBJ02456
      - CUR_OBJ02455
      - CUR_OBJ02457
      - CUR_OBJ02453
      - CUR_OBJ02473
      - CUR_OBJ02476
    - SCOPE_
    - FUT_

CUR_OBJ02454	PS_Project_type	calculate
!=NA	aan	NA
NA	!=aan	Constants.EMPTY_STRING

Figure 5.3: Calculate table example


UCS-2.4 × UCS-0000002457-1.0 × Sluishoofd-voorbeeld-1.4 ×

Sluishoofd-voorbeeld - 1.4 - work

filter ...

- Sluishoofd-voorbeeld
  - Sluishoofd-voorbeeld
    - PS_
    - ENV_
    - CUR_
      - CUR_OBJ02454
        - Horizontaal lierwerk
        - Verticaal lierwerk
        - Elektrohydraulische cilinder
        - Elektromechanische cilinder
        - NA
        - CUR_OBJ02454-visible-dt1
        - CUR_OBJ02454-option_visible-dt1
        - CUR_OBJ02454-calculate-dt1
        - CUR_OBJ02454-product-dt1
      - CUR_OBJ02456
      - CUR_OBJ02455
      - CUR_OBJ02457
      - CUR_OBJ02453
      - CUR_OBJ02473
      - CUR_OBJ02476
    - SCOPE_
    - FUT_

Edit Decision Table

Rows Columns Clipboard Export

CUR_OBJ02454	PS_Project_type	visible	
!=NA	vnr	✓	Edit

Figure 5.4: Visible table example

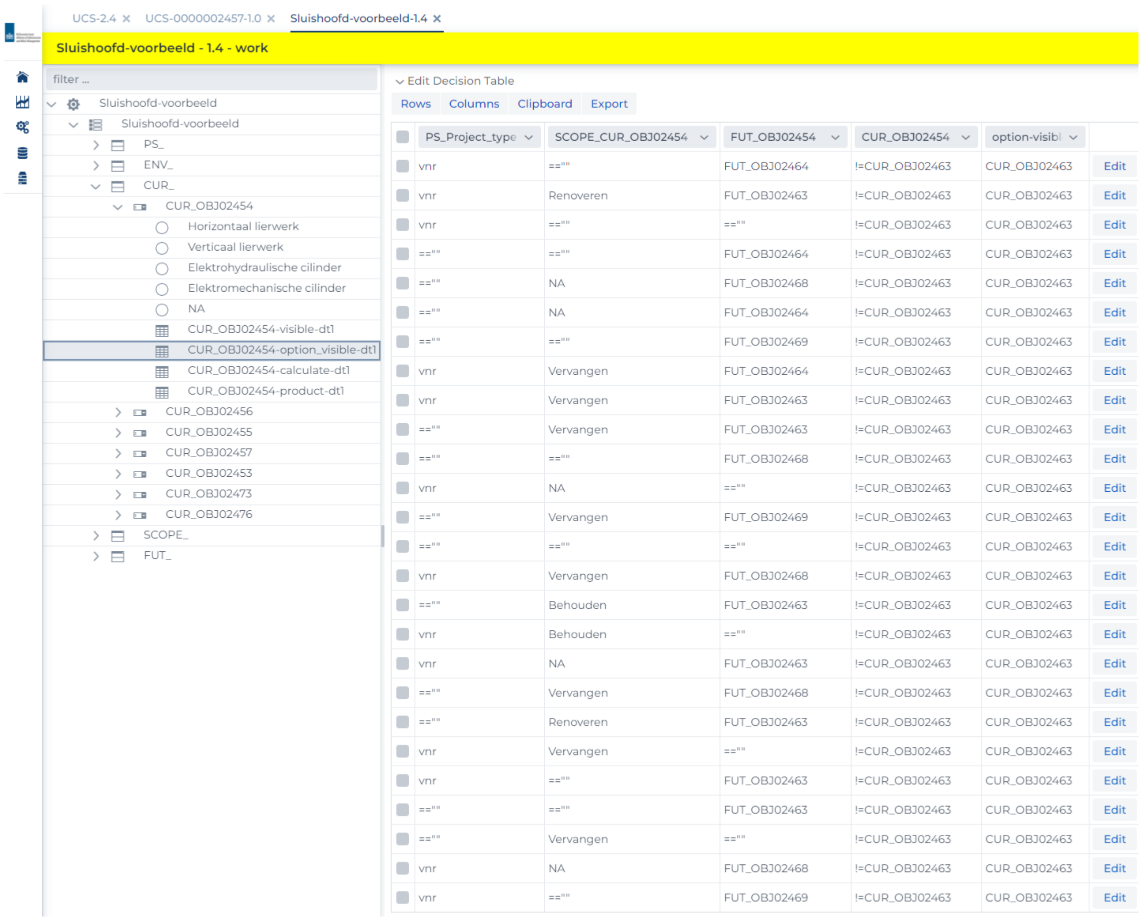
The generated XML-template can be imported into Merkato and used to configure products. Figures 5.8 and 5.9 illustrate the functioning of the configuration for greenfield and brownfield respectively.

5.4 Validation of configurator data structure

The configurator data structure is used to create a configurator that can only produce valid configurations accordingly. The configurator data structure is partly derived from the product platform and supplemented by hand with LSR and design rules by the RWS users. When supplementing the design rules, mistakes could be made, such as entering incorrect design rules or forgetting design rules. This shifts the human error from wrong configurator development to creating a wrong configurator data structure, which is less error prone. It is important to check the configurator data structure thoroughly, such that the configurator behaves as intended. It can be helpful to test the configurator in Merkato by making several configurations. In case a mistake is found, it should be corrected accordingly. E.g. remove, add or change the data in the configurator data structure. Then the configurator can be updated with the new data by regenerating the configurator using the Python script.

5.5 Generation of project requirements specification

The products on the quote list can be linked to a system specification from GRIP to eventually get the complete requirements specification for a lock configuration, as illustrated in Figure 5.10. Currently, only answers and the combination of the current situation and component scope are quoted as products. Connections between components are not included at the moment. The list of quoted products from a configuration determines which requirements specifications are needed


UCS-2.4 × UCS-0000002457-1.0 × Sluishoofd-voorbeeld-1.4 ×

Sluishoofd-voorbeeld - 1.4 - work

filter ...

- Sluishoofd-voorbeeld
  - Sluishoofd-voorbeeld
    - PS_
    - ENV_
    - CUR_
      - CUR_OBJ02454
        - Horizontaal lierwerk
        - Verticaal lierwerk
        - Elektrohydraulische cilinder
        - Elektromechanische cilinder
        - NA
        - CUR_OBJ02454-visible-dt1
        - CUR_OBJ02454-option_visible-dt1
        - CUR_OBJ02454-calculate-dt1
        - CUR_OBJ02454-product-dt1
      - CUR_OBJ02456
      - CUR_OBJ02455
      - CUR_OBJ02457
      - CUR_OBJ02453
      - CUR_OBJ02473
      - CUR_OBJ02476
    - SCOPE_
    - FUT_

Edit Decision Table

Rows Columns Clipboard Export

PS_Project_type	SCOPE_CUR_OBJ02454	FUT_OBJ02454	CUR_OBJ02454	option-visibl	
vnr	==""	FUT_OBJ02464	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	Renoveren	FUT_OBJ02463	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	==""	==""	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	==""	FUT_OBJ02464	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	NA	FUT_OBJ02468	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	NA	FUT_OBJ02464	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	==""	FUT_OBJ02469	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	Vervangen	FUT_OBJ02464	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	Vervangen	FUT_OBJ02463	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	Vervangen	FUT_OBJ02463	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	==""	FUT_OBJ02468	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	NA	==""	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	Vervangen	FUT_OBJ02469	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	==""	==""	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	Vervangen	FUT_OBJ02468	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	Behouden	FUT_OBJ02463	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	Behouden	==""	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	NA	FUT_OBJ02463	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	Vervangen	FUT_OBJ02468	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	Renoveren	FUT_OBJ02463	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	Vervangen	==""	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	==""	FUT_OBJ02463	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	==""	FUT_OBJ02463	!=CUR_OBJ02463	CUR_OBJ02463	Edit
==""	Vervangen	==""	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	NA	FUT_OBJ02468	!=CUR_OBJ02463	CUR_OBJ02463	Edit
vnr	==""	FUT_OBJ02469	!=CUR_OBJ02463	CUR_OBJ02463	Edit

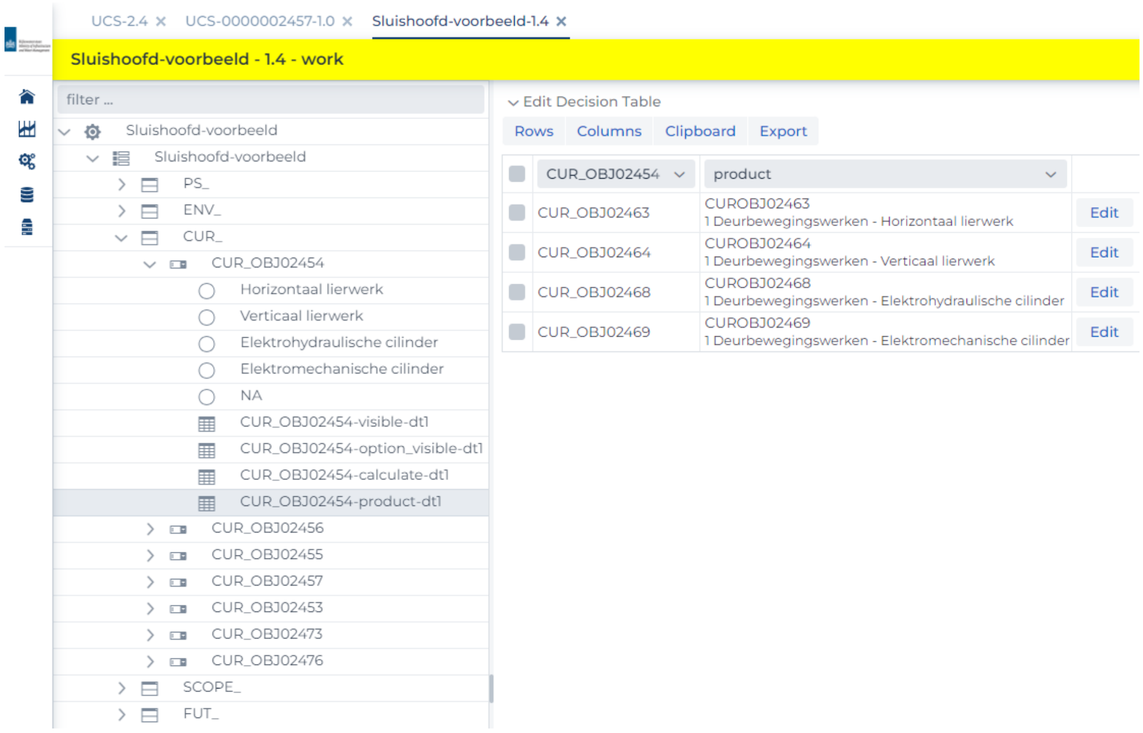
Figure 5.5: Option-visible table example

for a project. This list is illustrated for a greenfield and brownfield project respectively in Figures 5.11 and 5.12.

5.6 Results

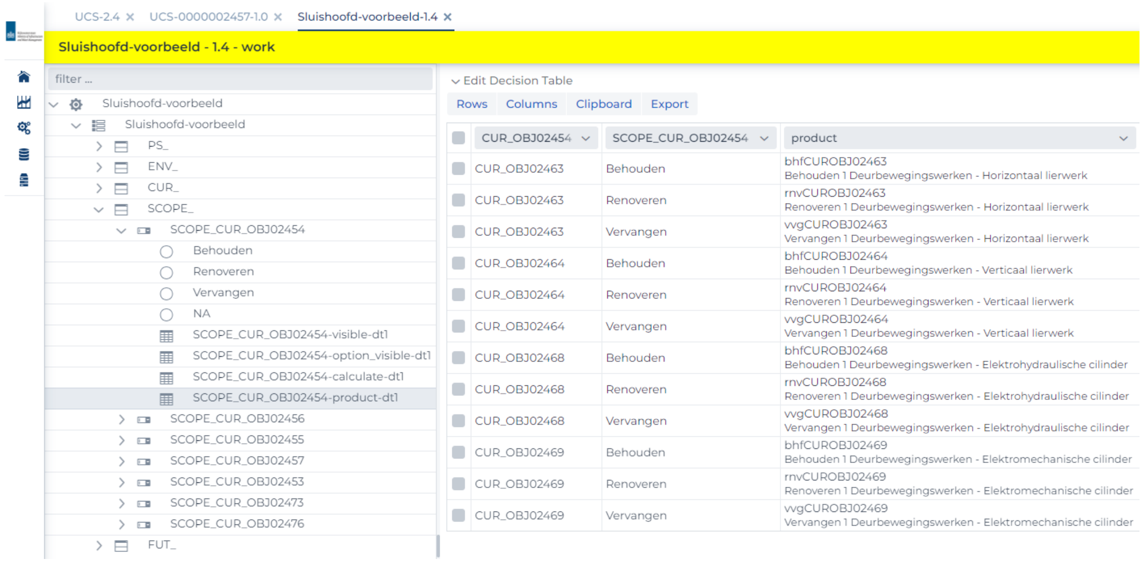
In this chapter, it is demonstrated that the developed method is applicable to locks. The method utilized a product platform, LSR, and design rules to create a configurator suitable for both greenfield and brownfield configurations. This chapter focuses on the development of a lock head configurator, which is a complex lock module. However, the extension to a complete lock configurator is also tested. The lock head configurator is generated in less than 10 seconds, and the configurator for the complete lock in approximately 1 minute. However, the design rules for the complete lock have not yet been defined.

Given the number of questions, answers, and answer-visible rules in the templates, the time savings compared to manual development is enormous. Additionally, because the configurator is synthesised, it guarantees that the configurator can only produce valid configurations, unlike manual development in which RWS users can make mistakes. The LSR and design rules can be added to the Python file, and the product platform can be maintained in GRIP and exported, simplifying the use and maintenance of the method. The configurator in Merkato is clear, user-friendly, and intuitive to use. It ensures the production of valid configurations and generates a list of necessary products for the created configuration, from which a requirements specification for the project can be made.



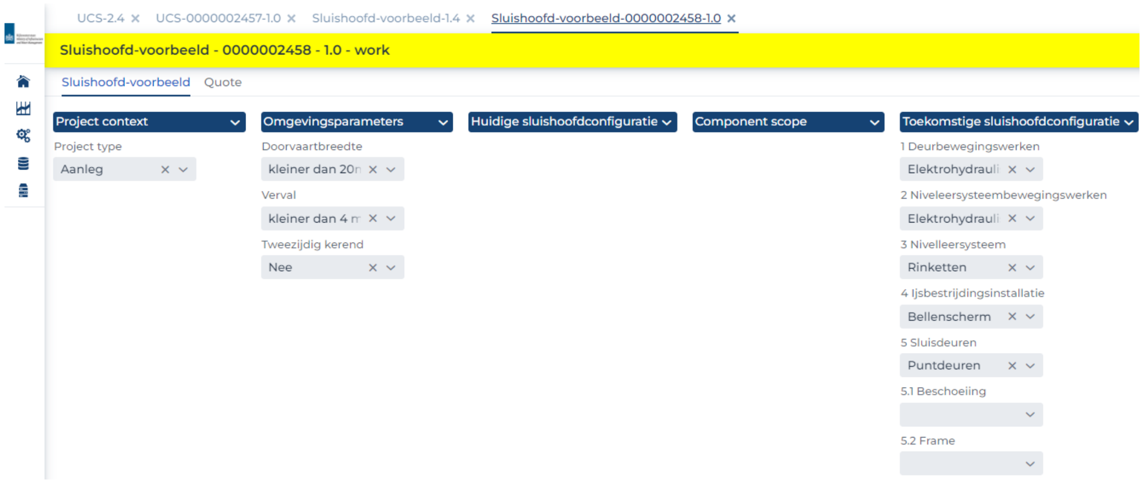
CUR_OBJ02454	product	
CUR_OBJ02463	CUROBJ02463 1 Deurbewegingswerken - Horizontaal lierwerk	Edit
CUR_OBJ02464	CUROBJ02464 1 Deurbewegingswerken - Verticaal lierwerk	Edit
CUR_OBJ02468	CUROBJ02468 1 Deurbewegingswerken - Elektrohydraulische cilinder	Edit
CUR_OBJ02469	CUROBJ02469 1 Deurbewegingswerken - Elektromechanische cilinder	Edit

Figure 5.6: Example of a product tabel with a single condition column



CUR_OBJ02454	SCOPE_CUR_OBJ02454	product
CUR_OBJ02463	Behouden	bhfCUROBJ02463 Behouden 1 Deurbewegingswerken - Horizontaal lierwerk
CUR_OBJ02463	Renoveren	rnvCUROBJ02463 Renoveren 1 Deurbewegingswerken - Horizontaal lierwerk
CUR_OBJ02463	Vervangen	vvgCUROBJ02463 Vervangen 1 Deurbewegingswerken - Horizontaal lierwerk
CUR_OBJ02464	Behouden	bhfCUROBJ02464 Behouden 1 Deurbewegingswerken - Verticaal lierwerk
CUR_OBJ02464	Renoveren	rnvCUROBJ02464 Renoveren 1 Deurbewegingswerken - Verticaal lierwerk
CUR_OBJ02464	Vervangen	vvgCUROBJ02464 Vervangen 1 Deurbewegingswerken - Verticaal lierwerk
CUR_OBJ02468	Behouden	bhfCUROBJ02468 Behouden 1 Deurbewegingswerken - Elektrohydraulische cilinder
CUR_OBJ02468	Renoveren	rnvCUROBJ02468 Renoveren 1 Deurbewegingswerken - Elektrohydraulische cilinder
CUR_OBJ02468	Vervangen	vvgCUROBJ02468 Vervangen 1 Deurbewegingswerken - Elektrohydraulische cilinder
CUR_OBJ02469	Behouden	bhfCUROBJ02469 Behouden 1 Deurbewegingswerken - Elektromechanische cilinder
CUR_OBJ02469	Renoveren	rnvCUROBJ02469 Renoveren 1 Deurbewegingswerken - Elektromechanische cilinder
CUR_OBJ02469	Vervangen	vvgCUROBJ02469 Vervangen 1 Deurbewegingswerken - Elektromechanische cilinder

Figure 5.7: Example of a product tabel with multiple condition columns



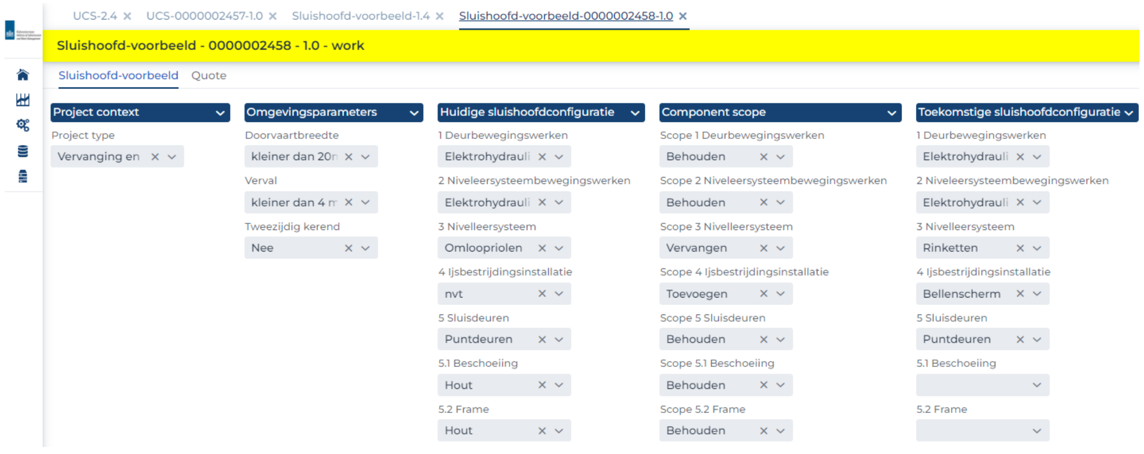
UCS-2.4 × UCS-0000002457-1.0 × Sluishoofd-voorbeeld-1.4 × Sluishoofd-voorbeeld-0000002458-1.0 ×

Sluishoofd-voorbeeld - 0000002458 - 1.0 - work

Sluishoofd-voorbeeld Quote

Project context	Omgevingsparameters	Huidige sluishoofdconfiguratie	Component scope	Toekomstige sluishoofdconfiguratie
Project type: Aanleg	Doorvaartbreedte: kleiner dan 20r			1 Deurbewegingswerken: Elektrohydrauli
	Verval: kleiner dan 4 m			2 Niveleersysteembewegingswerken: Elektrohydrauli
	Tweezijdig kerend: Nee			3 Nivelleersysteem: Rinketten
				4 Ijsbestrijdingsinstallatie: Bellenscherm
				5 Sluisdeuren: Puntdeuren
				5.1 Beschoeiing:
				5.2 Frame:

Figure 5.8: Lock head configurator in Merkato for greenfield



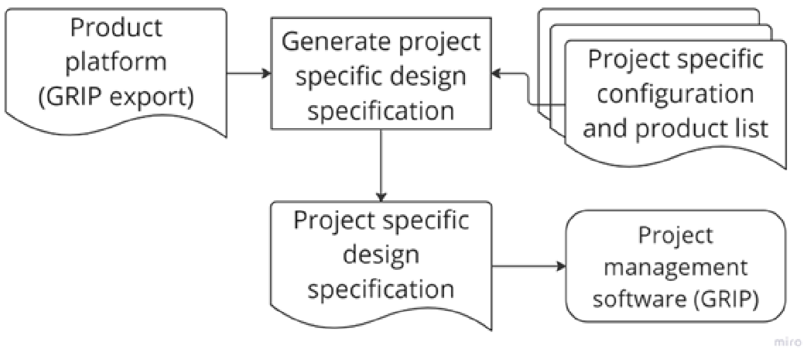
UCS-2.4 × UCS-0000002457-1.0 × Sluishoofd-voorbeeld-1.4 × Sluishoofd-voorbeeld-0000002458-1.0 ×

Sluishoofd-voorbeeld - 0000002458 - 1.0 - work

Sluishoofd-voorbeeld Quote

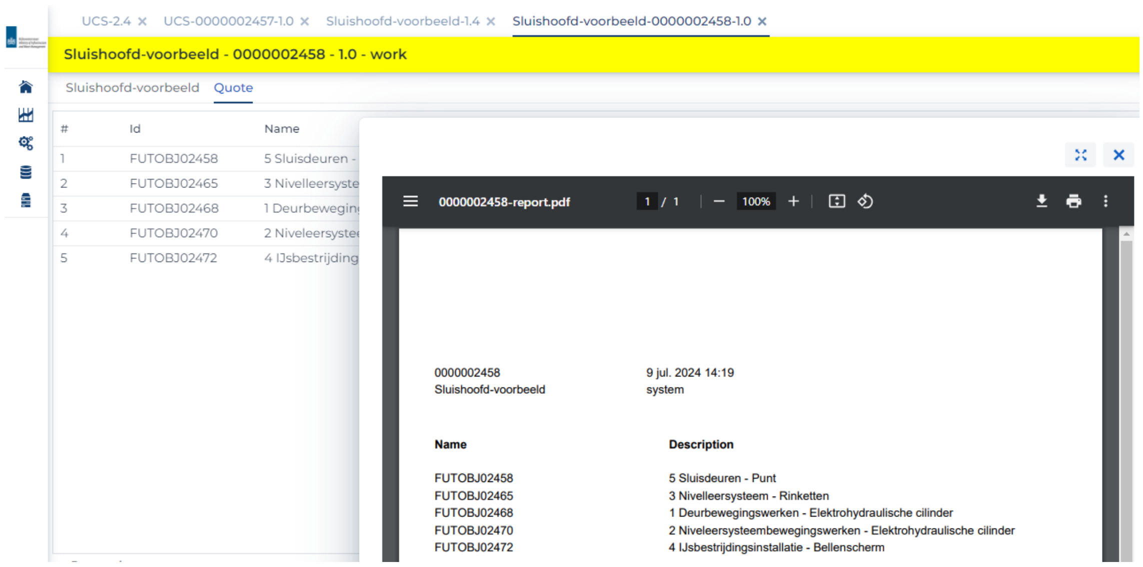
Project context	Omgevingsparameters	Huidige sluishoofdconfiguratie	Component scope	Toekomstige sluishoofdconfiguratie
Project type: Vervanging en	Doorvaartbreedte: kleiner dan 20r	1 Deurbewegingswerken: Elektrohydrauli	Scope 1 Deurbewegingswerken: Behouden	1 Deurbewegingswerken: Elektrohydrauli
	Verval: kleiner dan 4 m	2 Niveleersysteembewegingswerken: Elektrohydrauli	Scope 2 Niveleersysteembewegingswerken: Behouden	2 Niveleersysteembewegingswerken: Elektrohydrauli
	Tweezijdig kerend: Nee	3 Nivelleersysteem: Omloopriolen	Scope 3 Nivelleersysteem: Vervangen	3 Nivelleersysteem: Rinketten
		4 Ijsbestrijdingsinstallatie: nvt	Scope 4 Ijsbestrijdingsinstallatie: Toevoegen	4 Ijsbestrijdingsinstallatie: Bellenscherm
		5 Sluisdeuren: Puntdeuren	Scope 5 Sluisdeuren: Behouden	5 Sluisdeuren: Puntdeuren
		5.1 Beschoeiing: Hout	Scope 5.1 Beschoeiing: Behouden	5.1 Beschoeiing:
		5.2 Frame: Hout	Scope 5.2 Frame: Behouden	5.2 Frame:

Figure 5.9: Lock head configurator in Merkato for brownfield



Product platform (GRIP export)
Generate project specific design specification
Project specific configuration and product list
Project specific design specification
Project management software (GRIP)
miro

Figure 5.10: Generation of project requirements specification

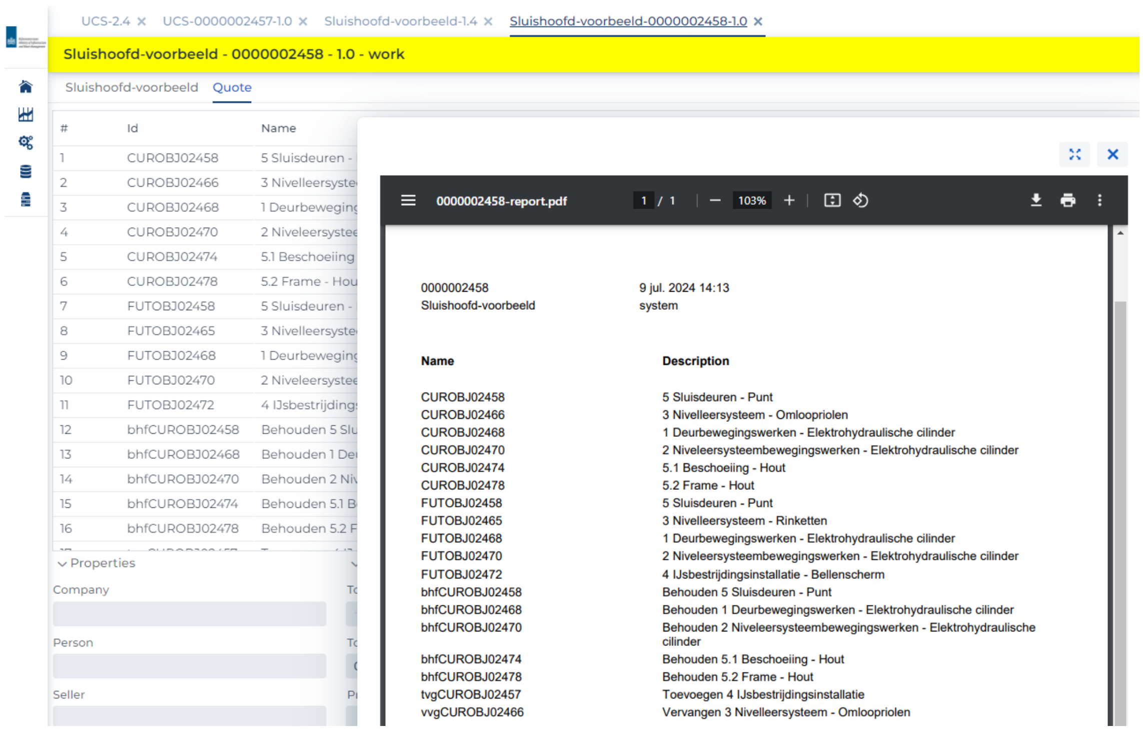

UCS-2.4 ×
UCS-0000002457-1.0 ×
Sluishoofd-voorbeeld-1.4 ×
Sluishoofd-voorbeeld-0000002458-1.0 ×
Sluishoofd-voorbeeld - 0000002458 - 1.0 - work
Sluishoofd-voorbeeld
Quote
0000002458-report.pdf
1 / 1
100%

#	Id	Name
1	FUTOBJ02458	5 Sluisdeuren -
2	FUTOBJ02465	3 Nivelleersyste
3	FUTOBJ02468	1 Deurbewegin
4	FUTOBJ02470	2 Niveleersyste
5	FUTOBJ02472	4 IJsbestrijding

0000002458 9 jul. 2024 14:19
Sluishoofd-voorbeeld system

Name	Description
FUTOBJ02458	5 Sluisdeuren - Punt
FUTOBJ02465	3 Nivelleersysteem - Rinketten
FUTOBJ02468	1 Deurbewegingswerken - Elektrohydraulische cilinder
FUTOBJ02470	2 Niveleersysteembewegingswerken - Elektrohydraulische cilinder
FUTOBJ02472	4 IJsbestrijdingsinstallatie - Bellenscherm

Figure 5.11: Products lock head for greenfield



#	Id	Name
1	CUROBJ02458	5 Sluisdeuren -
2	CUROBJ02466	3 Nivelleersyste
3	CUROBJ02468	1 Deurbeweging
4	CUROBJ02470	2 Niveleersystee
5	CUROBJ02474	5.1 Beschoeiing
6	CUROBJ02478	5.2 Frame - Hou
7	FUTOBJ02458	5 Sluisdeuren -
8	FUTOBJ02465	3 Nivelleersyste
9	FUTOBJ02468	1 Deurbeweging
10	FUTOBJ02470	2 Niveleersystee
11	FUTOBJ02472	4 IJsbestrijding
12	bhfCUROBJ02458	Behouden 5 Slu
13	bhfCUROBJ02468	Behouden 1 De
14	bhfCUROBJ02470	Behouden 2 Niv
15	bhfCUROBJ02474	Behouden 5.1 B
16	bhfCUROBJ02478	Behouden 5.2 F

0000002458 9 jul. 2024 14:13
Sluishoofd-voorbeeld system

Name	Description
CUROBJ02458	5 Sluisdeuren - Punt
CUROBJ02466	3 Nivelleersysteem - Omloopriolen
CUROBJ02468	1 Deurbewegingswerken - Elektrohydraulische cilinder
CUROBJ02470	2 Niveleersysteembewegingswerken - Elektrohydraulische cilinder
CUROBJ02474	5.1 Beschoeiing - Hout
CUROBJ02478	5.2 Frame - Hout
FUTOBJ02458	5 Sluisdeuren - Punt
FUTOBJ02465	3 Nivelleersysteem - Rinketten
FUTOBJ02468	1 Deurbewegingswerken - Elektrohydraulische cilinder
FUTOBJ02470	2 Niveleersysteembewegingswerken - Elektrohydraulische cilinder
FUTOBJ02472	4 IJsbestrijdingsinstallatie - Bellenscherm
bhfCUROBJ02458	Behouden 5 Sluisdeuren - Punt
bhfCUROBJ02468	Behouden 1 Deurbewegingswerken - Elektrohydraulische cilinder
bhfCUROBJ02470	Behouden 2 Niveleersysteembewegingswerken - Elektrohydraulische cilinder
bhfCUROBJ02474	Behouden 5.1 Beschoeiing - Hout
bhfCUROBJ02478	Behouden 5.2 Frame - Hout
tvgCUROBJ02457	Toevoegen 4 IJsbestrijdingsinstallatie
vvgCUROBJ02466	Vervangen 3 Nivelleersysteem - Omloopriolen

Figure 5.12: Products lock head for brownfield

Chapter 6

Discussion

The results from Chapter 5 show how a method utilizing Binary Decision Diagrams (BDD's) can be used to synthesize a rule-based product configurator. A configurator can be synthesized based on a (hierarchical) product platform, location specific requirements (LSR), and design rules. This approach ensures correctness by construction, guaranteeing that only valid configurations are produced. The method has been successfully tested by synthesising a lock head configurator. Furthermore, also a configurator for the complete lock has been synthesised, however this configurator lacks the necessary LSR and design rules. This chapter will discuss the results from the perspective of the research questions from Chapter 1. Additionally, this chapter addresses the limitations of the developed methodology and provides recommendations for its improvement.

6.1 How can the configurator solution space be defined?

The product platform is modeled as a data structure with questions, answers and follow-up questions. This configurator data structure is also used to specify other questions necessary in the configurator, e.g. the project type and LSR. Design rules are necessary to constrain the selection of variants. Three types of design rules are used and added to this configurator data structure, namely: exclude, enforce and precedence rules. From this configurator data structure, it is derived when an answer may be selected or not.

BDD's are based on the configurator data structure to determine if an answer may be selected or not. BDD's are convenient to use, because variables and constraints are added easily and can be evaluated efficiently. For each question, a BDD is made using only questions, answers and design rules relevant to that question. These small BDD's can be evaluated quickly preventing a memory overflow [36] [15]. In the BDD, variables are made for the question states and the possible events. These variables are constrained by Boolean functions derived from design rules (including the design rules derived from the hierarchy of the product platform). It is evaluated for which sets of variables the BDD's result in true, to determine if an event may take place. This is used to create an option-visible table which describes for each event under which conditions the event is allowed. This ensures that during configuration, the user is informed about permissible selections, guiding them through the process and preventing the selection of invalid answers.

Conventionally, the solution space is defined by creating answer-visible rules manually. However, many complex answer-visible rules are necessary, making it error-prone and labour intensive. Only for the lock head, the configurator requires 22136 rules, which are generated in less then 10 sections by using the synthesis method. This reduces implementation labor and costs, as well as minimizing costs from potential invalid configuration caused by human errors.

6.2 How can a configurator be developed to guarantee the production of valid configurations?

The requirements of the configurator are modelled in the configurator data structure, which is used to generate configurator rules. Three types of configurator rules are made: answer-visible, visible and calculate rules. The answer-visible rules determine if an answer may be selected based on a condition.

The generated answer-visible rules satisfy the provided design rules by definition due to the application of BDD's. Hence verification of the option-visible rules in the configurator is no longer needed. As a result, the configurator guarantees the production of valid configurations according the provided design rules. However, validation of the provided design rules is still needed. Validation is required, because design rules could be incomplete or wrong.

To guide the user through many questions in the configurator, it is chosen to only show the questions that need to be answered or have already been answered. This is facilitated using visible rules, ensuring a top-down selection through the product platform. Additionally, self-added precedence rules can be implemented, where specific questions become visible only after another question has been answered.

6.3 How can a configurator be used for both greenfield and brownfield projects?

For greenfield projects, the selection of the future situation depends on the LSR and design rules. However for brownfield projects, the current situation and component scope impose additional restrictions on what can be chosen in the future situation. By using a product platform to describe the current situation, component scope, future situation, and additional constraints, it is shown that a configurator can be applicable for both greenfield and brownfield configurations. The additional constraints are defined and automatically derived restricts the use of components in the future situation depending on the current situation and component scope.

The specification of the current and future configuration can only be done according to the product platform. However, RWS also has, for example, monumental locks of which the components and variants may not all have been added to the product platform, making the configurator unusable for them. This can be resolved by adding monumental components to the product platform and add design rules that only allow monumental components in the future situation if they are present in the current situation.

6.4 What approach can be utilized to facilitate configurator management and maintenance?

Management and maintenance of the configurator are focused on the product platform, design rules and template generation. That is, the product platform and design rules are expected to change over time. For example, due to developments in technology. Template generation is Merkato specific, as a consequence updates of Merkato might influence the template generation script.

The configurator itself does not need to be managed or maintained as it can be easily re-synthesised following the updates of the product platform, design rules and Merkato. This is a significant advantage over the conventional method, where the configurator must be checked and adjusted as needed, which is labour intensive and error prone.

This current research for RWS has resulted in an effective method for automatically developing a configurator for complex products. The key points include modeling the interactive configuration problem as a set of independent BDD's for each question, allowing intermediate possible selections to be evaluated, guaranteeing valid configurations according to the user guide, instead of calculating the complete solution space. Furthermore, the solution space can be defined by the constraints. Additionally, due to the smaller subproblems, the calculation time scales more linearly (rather than exponentially) with the number of questions and answers. The method can also be used to create configurators for both greenfield and brownfield projects. For brownfield projects, additional questions and answers are automatically derived from the product platform, and constraints are also automatically calculated. The challenges described in the problem statement have been successfully addressed.

6.5 Limitations

The method is applicable regardless of the content of the product platform and LSR. This allows the method to be used for synthesizing configurators for various products, such as bridges and tunnels. However, the current implementation is limited to the use of discrete answers. Not all questions in a product configurator are necessarily discrete, but can often be discretized in context of standardisation. The lock head width for example, could be any numerical value, however is discretized via step sizes, as shown in Table 2.1. Furthermore in context of standardisation, a company would rather use standardised dimensions (discrete) than produce any products of any size.

In Section 1.1, it is explained that the objective is to reduce sensitivity to human error. This is achieved by automatically generating the configurator, ensuring correctness through construction. Nevertheless, if there are errors in the configurator data structure, such as excluding an allowed combination, it still leads to human errors.

The scalability of the current method needs to be further investigated, as in theory one could have many design rules that link many question, resulting in large subproblem solution spaces, which will deteriorate computation of performance. It is currently unknown whether this is a theoretical issue or a practical issue as well. However, if scalability is problematic, one could resort to formulating BDD subproblems per answer. In the current application on the lock head, which is the most complex part of a lock, no scalability issues occurred.

Chapter 7

Conclusion and future work

7.1 Conclusion

RWS faces a significant challenge in renovating and replacing locks. The current Engineer-to-Order (EtO) production strategy has led to a large variety of locks in the product portfolio resulting in negative side effects. RWS wants to increase the degree of standardization by transitioning from an EtO to a Configure-to-Order (CtO) production strategy. To support the CtO strategy, a product platform is used and implemented as a product configurator. The traditional manual development of a configurator is error-prone and labour intensive.

In this thesis, a method is proposed for the automatic generation of a product configurator. This eliminates the need for manual design of the configurator. Furthermore, because the configurator is synthesized and thus correct by construction, it no longer requires verification, thereby reducing both error-proneness and labor intensity.

The product platform, along with the location specific requirements (LSR) and design rules, are necessary inputs for synthesizing the configurator. Within the configurator, three types of rules are required: answer-visible rules, visible rules, and calculate rules. Answer-visible rules are computed using Binary Decision Diagrams (BDD's) and ensure only valid configurations are produced. Binary variables for states and events, along with Boolean functions derived from the input, are added to the BDD. Visible rules serve to guide the user, while calculation rules are technically essential. By adjusting the configurator data structure, it can be utilized for both greenfield and brownfield projects.

A lock head configurator has been generated that RWS can use to create both greenfield and brownfield lock configurations. This configurator is tested and demonstrated in Merkato configurator software, which provides essential features and user-friendliness. All produced configurations are valid and result in a list of products to which requirements can be coupled, allowing for the creation of a requirements specification. Using the product platform and the configurator, RWS can transition to a CtO production strategy, leading to more requirements specifications and eventually to more standardized locks.

Many companies benefit from using a CtO production strategy, which often makes use of a product platform. This method can be used to implement such a product platform via a configurator. The method is generic and can therefore be applied to generate a product configurator for various products. As long as the data structure of the configurator is well-defined, a valid configurator will be produced. In collaboration with Quootz B.V., research will be conducted in the future to identify which companies could benefit from this method.

7.2 Future work

The method can be applied to the entire lock system and has already been successfully tested. However, work needs to be done on determining the LSR that influence the lock configuration and figuring out design rules. Also there may be various configurations possible, given the project type and LSR. Now, the user has the freedom to choose what seems best to them, however, in the context of standardization, it is questionable whether this is desired or whether preferred configurations or variants should be proposed where possible.

Additionally, the current requirement specifications are more at a conceptual level of what needs to be built; they do not prescribe dimensions or standardized components. This is because of the level of detail of the product platform. It might be beneficial to further elaborate on the product platform in detail to derive even more benefits from standardization. Furthermore, as mentioned in the discussion, the configurator does currently not support monumental components, because they are not in the product platform. Adding them to the product platform would resolve this

The products on the quote list refer to necessary requirement specifications. Currently, only products are put on the quote list for single variants. However, RWS also has requirement specifications for variant combinations. Future research can look into the calculation of product rules for variant combinations, so that requirement specifications for variant combinations can also be calculated in Merkato. Hence, based on the product list, the requirements specification document can be automatically generated in Merkato.

Lastly, the constructed rules and template are correct by construction, so do not have to be verified. However the configurator data structure should be validated by testing the generated configurator. Future work can look into the calculation of the solution space and the visualisation of the solution space.

Bibliography

[1] Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978. `doi:10.1109/TC.1978.1675141`.

[2] Paul Blažek, Monika Kolb, Clarissa Streichsbier, and Simone Honetz. *The Evolutionary Process of Product Configurators*. 2016. `doi:10.1007/978-3-319-29058-4\_13`.

[3] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, DAC '90, page 40–45, New York, NY, USA, 1991. Association for Computing Machinery. `doi:10.1145/123186.123222`.

[4] Edward Brown, Bala Chidambaram, and Gordon Aaseng. *Applying Health Management Technology to the NASA Exploration System-of-Systems*. 2005. `doi:10.2514/6.2005-6624`.

[5] Quootz B.V. Quootz. URL: `https://quootz.nl/`.

[6] Caltech Control and Dynamical Systems. Project description. URL: `https://pypi.org/project/dd/`.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery. `doi:10.1145/800157.805047`.

[8] Rolf Drechsler and Bernd Becker. *Binary decision diagrams*. Springer Science Business Media, 2013.

[9] Eclipse ESCET. Supervisory controllers, 2024. Accessed: 2024-05-23. URL: `https://eclipse.dev/escet/cif/synthesis-based-engineering/supervisory-controllers.html`.

[10] S.J. Friedman and K.J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *I.E.E.E. transactions on computers/IEEE transactions on computers*, 39(5):710–713, May 1990. `doi:10.1109/12.53586`.

[11] Esben Rune Hansen. *Decision Support using Finite Automata and Decision Diagrams*. PhD thesis, IT University of Copenhagen, 2008.

[12] Esben Rune Hansen and Henrik Reif Andersen. Interactive configuration with regular string constraints. *National Conference on Artificial Intelligence*, pages 217–223, 2007. URL: `https://aaaipress.org/Papers/AAAI/2007/AAAI07-033.pdf`.

[13] Anders Haug. *Representation of Industrial Knowledge - as a Basis for Developing and Maintaning Product Configurators*. PhD thesis, juli 2008.

[14] Anders Haug, Lars Hvam, and Niels Henrik Mortensen. Definition and evaluation of product configurator development strategies. *Computers in industry*, 63(5):471–481, June 2012. `doi:10.1016/j.compind.2012.02.001`.

[15] Peter Hawkins, Vitaly Lagoon, and Peter J. Stuckey. Solving Set Constraint Satisfaction Problems using ROBDDs. *Journal of artificial intelligence research/The journal of artificial intelligence research*, 24:109–156, 2005. `doi:10.1613/jair.1638`.

[16] Holger H. Hoos and Thomas Stützle. *Stochastic local search*. Morgan Kaufmann, 2005.

[17] Lars Hvam, Anders Haug, Niels Henrik Mortensen, and Christian Thuesen. Observed benefits from product configuration systems. *International Journal of Industrial Engineering (Online)*, 20(5-6):329–338, 2013.

[18] Sjoerd Carné Marjos Knippenberg. *Designing a product platform for ship locks*. PhD thesis, September 2023. Proefschrift.

[19] Katrin Kristjansdottir, Sara Shafiee, Lars Hvam, Martin Bonev, and Anna Myrodia. Quantification of benefits and cost from applying a product configuration system. 2016. 7¡sup¿th¡/sup¿ international conference on mass customization and personalization in Central Europe, MCP-CE2016 ; Conference date: 21-09-2016 Through 23-09-2016. URL: `http://www.mcp-ce.org/`.

[20] Harry Lammeretz. Plan van aanpak multiwaterwerk, 2021. Intern document.

[21] Marc H. Meyer and Alvin P. Lehnerd. *The power of product platforms: building value and cost leadership*. 1997. URL: `http://ci.nii.ac.jp/ncid/BA29825852`.

[22] Sajed Miremadi, Bengt Lennartson, and Knut Akesson. A bdd-based approach for modeling plant and supervisor by extended finite automata. *IEEE transactions on control systems technology*, 20(6):1421–1435, 2012. `doi:10.1109/tcst.2011.2167150`.

[23] Moraneus. Navigating the intricacies of binary decision diagrams (bdds), 2023. Accessed: 2024-05-26. URL: `https://medium.com/@moraneus/navigating-the-intricacies-of-binary-decision-diagrams-bdds-4273d6f6df42`.

[24] W.J. Pennings. Development of a lock head product platform. Master's thesis, Eindhoven University of Technology, 2021. URL: `https://pure.tue.nl/ws/portalfiles/portal/174405743/0954902_Pennings.pdf`.

[25] Qlic. Wat is xml, 2024. Accessed: 2024-06-04. URL: `https://www.qlic.nl/kennisbank/xml/155/#:~:text=XML%2C%20de%20afkorting%20voor%20Extensible,platte%20tekst%20kunnen%20worden%20weergegeven`.

[26] F.F.H. Reijnen, M.A. Goorden, J.M. Van De Mortel-Fronczak, and J.E. Rooda. Supervisory control synthesis for a waterway lock. *2017 IEEE Conference on Control Technology and Applications (CCTA)*, 2017. `doi:10.1109/ccta.2017.8062679`.

[27] Rijkswaterstaat. Dammen, sluizen en stuwen. `https://www.rijkswaterstaat.nl/water/waterbeheer/bescherming-tegen-het-water/waterkeringen/dammen-sluizen-enstuwen`.

[28] Rijkswaterstaat. About us. `https://www.rijkswaterstaat.nl/en/about-us`, n.d.

[29] Rijkswaterstaat. Main waterway network. `https://www.rijkswaterstaat.nl/en/water/water-management/main-waterway-network`, n.d.

[30] Radboud University Software Science Group. Synthesis based engineering, 2024. Accessed: 2024-05-23. URL: `https://sws.cs.ru.nl/Teaching/SynthesisBasedEngineering`.

[31] Sander Thuijsman and Michel Reniers. Supervisory control for dynamic feature configuration in product lines. 2023. `doi:10.1145/3579644`.

[32] Tatsuhiro Tsuchiya. Using binary decision diagrams for constraint handling in combinatorial interaction testing, 2019. `arXiv:1907.01779`.

[33] E.M.J. van Rijsbergen. Configurable requirements specification for platform-based renovation of ship locks. Master's thesis, Eindhoven University of Technology, 2024.

[34] E Vareilles, Christian Thuesen, Marie Falcon, and Michel Aldanondo. Interactive configuration of high performance renovation of apartment buildings by the use of csp. pages 29–34, 2013.

[35] Mathieu Veron and Michel Aldanondo. Yet another approach to ccsp for configuration problem. pages 59–62, 2000.

[36] Alexey Voronov. *On Formal Methods for Large-Scale Product Configuration*. PhD thesis, Chalmers university of technology, 2013. URL: `https://publications.lib.chalmers.se/records/fulltext/168379/168379.pdf`.

[37] T. Wilschut, L. F. P. Etman, J. E. Rooda, and J. A. Vogel. Similarity, modularity, and commonality analysis of navigation locks in the netherlands. *Journal of infrastructure systems*, 25(1), 2019. `doi:10.1061/(asce)is.1943-555x.0000468`.

[38] Tim Wilschut. *System specification and design structuring methods for a lock product platform*. PhD thesis, Eindhoven University of Technology, November 2018. Proefschrift.

[39] Tim Wilschut. Multiwaterwerk – van onderzoek naar toepassing, 2022. Intern document.

[40] Tim Wilschut. Begeleidend schrijven productplatform schutsluizen in grip, 2023. Intern document.

[41] Tim Wilschut. Managing product variety, 2023. PowerPoint slides, Rijkswaterstaat, Utrecht, Nederland.

[42] Linda L. Zhang, Petri T. Helo, Arun Kumar, and Xiao You. Implications of product configurator applications: An empirical study. pages 57–61, 2015. `doi:10.1109/IEEM.2015.7385608`.

Appendix A

Product platform

A concept of the product platform is displayed in Figure A.1. This product platform has 8 levels and does not contain all possible components and variants. However, it sketches the basic structure of the tree.

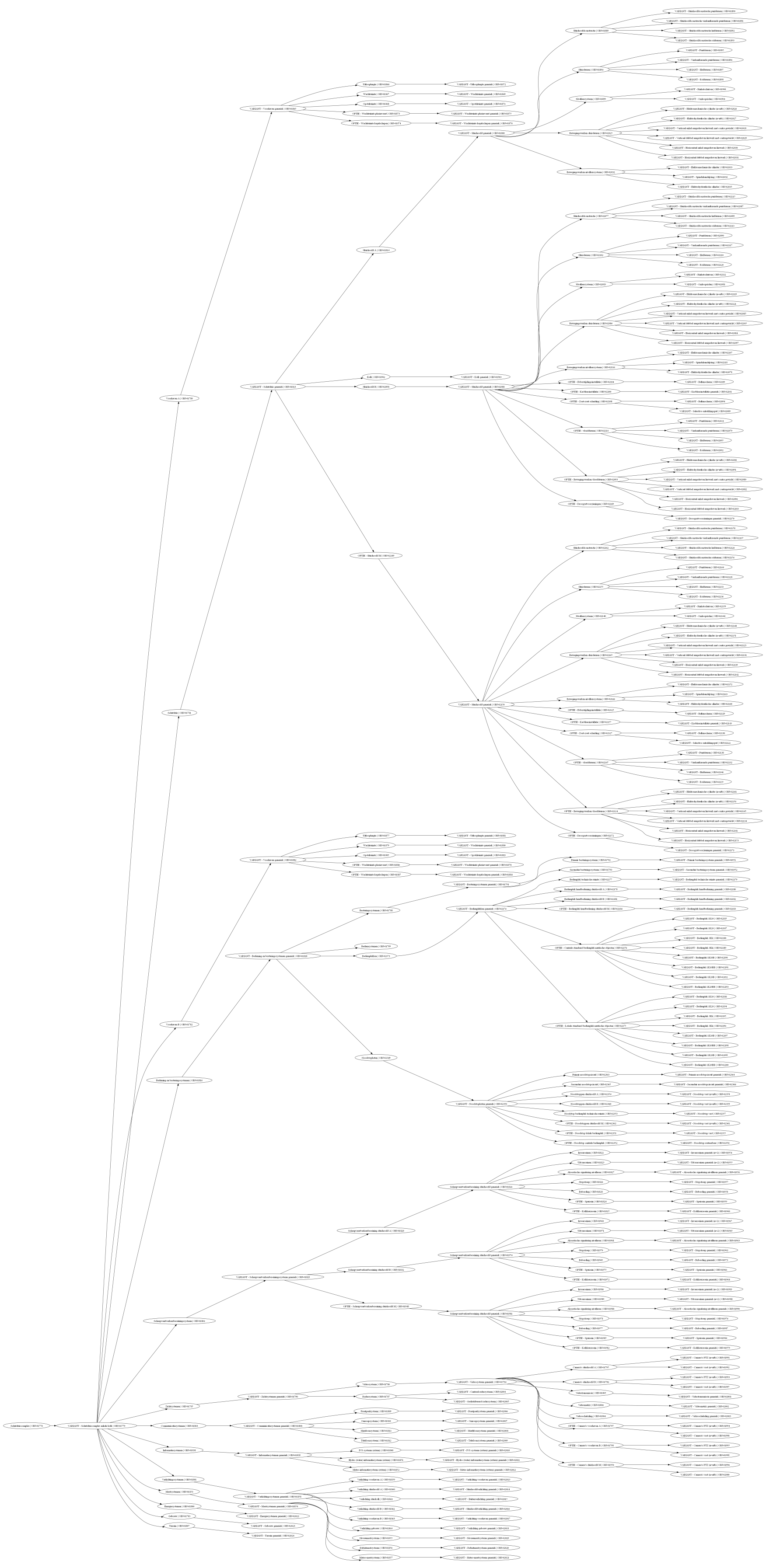


Figure A.1: Product platform

Appendix B

Configurator data structure for greenfield and brownfield

Listing B.1: Part of lockhead configurator data

```
cf_method2 = {
    "PS_Project_type": {
        "greenfield": [
            "FUT_A",
            "ENV_J"
        ],
        "brownfield": [
            "CUR_A",
            "FUT_A",
            "ENV_J",
        ]
    },
    "ENV_J": {
        "ENV_J1": []
    },
    "CUR_A": {
        "CUR_A1": ["CUR_B", "CUR_C", "SCOPE_CUR_A"],
        "CUR_A2": ["CUR_D", "CUR_E", "SCOPE_CUR_A"]
    },
    "CUR_B": {
        "CUR_B1": ["CUR_F", "CUR_G", "SCOPE_CUR_B"],
        "CUR_B2": ["SCOPE_CUR_B"]
    },
    "CUR_C": {
        "CUR_C1": ["CUR_H", "SCOPE_CUR_C"],
        "CUR_C2": ["SCOPE_CUR_C"]
    },
    "CUR_D": {
        "CUR_D1": ["CUR_I", "SCOPE_CUR_D"],
        "CUR_D2": ["CUR_I", "SCOPE_CUR_D"],
        "CUR_D3": ["SCOPE_CUR_D"]
    },
    "CUR_E": {
        "CUR_E1": ["SCOPE_CUR_E"]
    },
    "CUR_F": {
        "CUR_F1": ["SCOPE_CUR_F"],
        ”CUR_F2”: [”SCOPE_CUR_F”]
    },
    ”CUR_G”: {
        ”CUR_G1”: [”SCOPE_CUR_G”],
        ”CUR_G2”: [”SCOPE_CUR_G”]
    },
    ”CUR_H”: {
        ”CUR_H1”: [”SCOPE_CUR_H”],
        ”CUR_H2”: [”SCOPE_CUR_H”]
    },
    ”CUR_I”: {
        ”CUR_I1”: [”SCOPE_CUR_I”],
        ”CUR_I2”: [”SCOPE_CUR_I”]
    },
    ”SCOPE_CUR_A”: {
        ”Retain”: [],
        ”Renovate”: [],
        ”Replace”: []
    },
    ”SCOPE_CUR_B”: {
        ”Retain”: [],
        ”Renovate”: [],
        ”Replace”: []
    },
    ”SCOPE_CUR_C”: {
        ”Retain”: [],
        ”Renovate”: [],
        ”Replace”: []
    },
    ”SCOPE_CUR_D”: {
        ”Retain”: [],
        ”Renovate”: [],
        ”Replace”: []
    },
    ”SCOPE_CUR_E”: {
        ”Retain”: [],
        ”Renovate”: [],
        ”Replace”: []
    },
    ”SCOPE_CUR_F”: {
        ”Retain”: [],
        ”Renovate”: [],
        ”Replace”: []
    },
    ”SCOPE_CUR_G”: {
        ”Retain”: [],
        ”Renovate”: [],
        ”Replace”: []
    },
    ”SCOPE_CUR_H”: {
        ”Retain”: [],
        ”Renovate”: [],
        ”Replace”: []
    },
    ”SCOPE_CUR_I”: {
        ”Retain”: [],
        "Renovate": [],
        "Replace": []
    },
    "FUT_A": {
        "FUT_A1": ["FUT_B", "FUT_C"],
        "FUT_A2": ["FUT_D", "FUT_E"]
    },
    "FUT_B": {
        "FUT_B1": ["FUT_F", "FUT_G"],
        "FUT_B2": []
    },
    "FUT_C": {
        "FUT_C1": ["FUT_H"],
        "FUT_C2": []
    },
    "FUT_D": {
        "FUT_D1": ["FUT_I"],
        "FUT_D2": ["FUT_I"],
        "FUT_D3": []
    },
    "FUT_E": {
        "FUT_E1": []
    },
    "FUT_F": {
        "FUT_F1": [],
        "FUT_F2": []
    },
    "FUT_G": {
        "FUT_G1": [],
        "FUT_G2": []
    },
    "FUT_H": {
        "FUT_H1": [],
        "FUT_H2": []
    },
    "FUT_I": {
        "FUT_I1": [],
        "FUT_I2": []
    }
}

enforces_method2 = [[("FUT_D", "FUT_D1"), ("FUT_I", "FUT_I1")]]

excludes_method2 = [
    [("FUT_F", "FUT_F1"), ("FUT_G", "FUT_G1")],
    [("FUT_B", "FUT_B2"), ("FUT_H", "FUT_H1"), ("ENV_J", "ENV_J1")],
    [("CUR_A", "CUR_A1"), ("SCOPE_CUR_A", "Renovate"), ("FUT_A", "FUT_A2")],
    [("CUR_A", "CUR_A2"), ("SCOPE_CUR_A", "Renovate"), ("FUT_A", "FUT_A1")],
    [("CUR_B", "CUR_B1"), ("SCOPE_CUR_B", "Renovate"), ("FUT_B", "FUT_B2")],
    [("CUR_B", "CUR_B2"), ("SCOPE_CUR_B", "Renovate"), ("FUT_B", "FUT_B1")],
    [("CUR_C", "CUR_C1"), ("SCOPE_CUR_C", "Renovate"), ("FUT_C", "FUT_C2")],
    [("CUR_C", "CUR_C2"), ("SCOPE_CUR_C", "Renovate"), ("FUT_C", "FUT_C1")],
    [("CUR_D", "CUR_D1"), ("SCOPE_CUR_D", "Renovate"), ("FUT_D", "FUT_D2")],
    [("CUR_D", "CUR_D1"), ("SCOPE_CUR_D", "Renovate"), ("FUT_D", "FUT_D3")],
    [("CUR_D", "CUR_D2"), ("SCOPE_CUR_D", "Renovate"), ("FUT_D", "FUT_D1")],
    [("CUR_D", "CUR_D2"), ("SCOPE_CUR_D", "Renovate"), ("FUT_D", "FUT_D3")],
    [("CUR_D", "CUR_D3"), ("SCOPE_CUR_D", "Renovate"), ("FUT_D", "FUT_D1")],
    [("CUR_D", "CUR_D3"), ("SCOPE_CUR_D", "Renovate"), ("FUT_D", "FUT_D2")],
    [("CUR_F", "CUR_F1"), ("SCOPE_CUR_F", "Renovate"), ("FUT_F", "FUT_F2")],
    [("CUR_F", "CUR_F2"), ("SCOPE_CUR_F", "Renovate"), ("FUT_F", "FUT_F1")],
    [("CUR_G", "CUR_G1"), ("SCOPE_CUR_G", "Renovate"), ("FUT_G", "FUT_G2")],
    [("CUR_G", "CUR_G2"), ("SCOPE_CUR_G", "Renovate"), ("FUT_G", "FUT_G1")],
    [("CUR_H", "CUR_H1"), ("SCOPE_CUR_H", "Renovate"), ("FUT_H", "FUT_H2")],
    [("CUR_H", "CUR_H2"), ("SCOPE_CUR_H", "Renovate"), ("FUT_H", "FUT_H1")],
    [("CUR_I", "CUR_I1"), ("SCOPE_CUR_I", "Renovate"), ("FUT_I", "FUT_I2")],
    [("CUR_I", "CUR_I2"), ("SCOPE_CUR_I", "Renovate"), ("FUT_I", "FUT_I1")],
    [("CUR_A", "CUR_A1"), ("SCOPE_CUR_A", "Retain"), ("FUT_A", "FUT_A2")],
    [("CUR_A", "CUR_A2"), ("SCOPE_CUR_A", "Retain"), ("FUT_A", "FUT_A1")],
    [("CUR_B", "CUR_B1"), ("SCOPE_CUR_B", "Retain"), ("FUT_B", "FUT_B2")],
    [("CUR_B", "CUR_B2"), ("SCOPE_CUR_B", "Retain"), ("FUT_B", "FUT_B1")],
    [("CUR_C", "CUR_C1"), ("SCOPE_CUR_C", "Retain"), ("FUT_C", "FUT_C2")],
    [("CUR_C", "CUR_C2"), ("SCOPE_CUR_C", "Retain"), ("FUT_C", "FUT_C1")],
    [("CUR_D", "CUR_D1"), ("SCOPE_CUR_D", "Retain"), ("FUT_D", "FUT_D2")],
    [("CUR_D", "CUR_D1"), ("SCOPE_CUR_D", "Retain"), ("FUT_D", "FUT_D3")],
    [("CUR_D", "CUR_D2"), ("SCOPE_CUR_D", "Retain"), ("FUT_D", "FUT_D1")],
    [("CUR_D", "CUR_D2"), ("SCOPE_CUR_D", "Retain"), ("FUT_D", "FUT_D3")],
    [("CUR_D", "CUR_D3"), ("SCOPE_CUR_D", "Retain"), ("FUT_D", "FUT_D1")],
    [("CUR_D", "CUR_D3"), ("SCOPE_CUR_D", "Retain"), ("FUT_D", "FUT_D2")],
    [("CUR_F", "CUR_F1"), ("SCOPE_CUR_F", "Retain"), ("FUT_F", "FUT_F2")],
    [("CUR_F", "CUR_F2"), ("SCOPE_CUR_F", "Retain"), ("FUT_F", "FUT_F1")],
    [("CUR_G", "CUR_G1"), ("SCOPE_CUR_G", "Retain"), ("FUT_G", "FUT_G2")],
    [("CUR_G", "CUR_G2"), ("SCOPE_CUR_G", "Retain"), ("FUT_G", "FUT_G1")],
    [("CUR_H", "CUR_H1"), ("SCOPE_CUR_H", "Retain"), ("FUT_H", "FUT_H2")],
    [("CUR_H", "CUR_H2"), ("SCOPE_CUR_H", "Retain"), ("FUT_H", "FUT_H1")],
    [("CUR_I", "CUR_I1"), ("SCOPE_CUR_I", "Retain"), ("FUT_I", "FUT_I2")],
    [("CUR_I", "CUR_I2"), ("SCOPE_CUR_I", "Retain"), ("FUT_I", "FUT_I1")]
]

precedence_method2 = {
    "ENV_J": ["CUR_A", "FUT_A"],
    "CUR_F": ["CUR_G"],
    "FUT_F": ["FUT_G"],
    "SCOPE_CUR_A": ["FUT_A"],
    "SCOPE_CUR_B": ["FUT_B"],
    "SCOPE_CUR_C": ["FUT_C"],
    "SCOPE_CUR_D": ["FUT_D"],
    "SCOPE_CUR_E": ["FUT_E"],
    "SCOPE_CUR_F": ["FUT_F"],
    "SCOPE_CUR_G": ["FUT_G"],
    "SCOPE_CUR_H": ["FUT_H"],
    "SCOPE_CUR_I": ["FUT_I"]
}
```

Appendix C

Python scripts

The Python scripts created for this thesis are handed in separately. In total there are five scripts, each with its own purpose. The scripts are briefly explained below.:

- grip2cf.ipynb: this script is used to create the configurator data structure. For this, it imports and uses functions from "configuration_trees.py". Moreover, this script can also be used to parse a product platform export from GRIP into a configurator data structure, with the possibility to add design rules manually;
- configuration_trees.py: this script provides functions to "grip2cf.ipynb";
- cf2merkato.ipynb: this script is main file of the method. This script is used to generate the general configurator rules and then creates an Merkato configurator file in XML-format that is to be imported into Merkato. For this, it imports and uses functions from "configurator.py" and "merkato.py";
- configurator.py: this script contains general functions used in the method to create the configurator rules;
- merkato.py: this script contains Merkato-specific functions used to create the Merkato configurator template.