Constraint Modelling and Data Validation Using Formal Specification Languages

Inaugural-Dissertation

zur
Erlangung des Doktorgrades der
Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

David Schneider

aus Frankfurt am Main

Düsseldorf, Dezember 2017

Aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Referent:	Prof. Dr. Michael Leuschel Heinrich-Heine-Universität Düsseldorf
1. Koreferent:	Prof. Dr. Yves Ledru Université Grenoble Alpes

Tag der mündlichen Prüfung: 12. Juli 2017

Abstract

Formal methods provide rich and expressive specification languages to reason about and to describe systems at a high abstraction level. These methods are usually supported by powerful tools to verify the correctness of the specifications by different means such as proof or model checking. But is it possible to express non-trivial constraint satisfaction problems in these specification languages and to use such a formal model at runtime for problem solving and data validation? Are languages and the tools powerful enough to enable this usage scenario? These are the central questions we explore in this thesis. We look at these questions with a particular focus on the B Method – a state based formal method for software development – and the ProB tool – an animator and model checker for the B Method.
We begin by studying the use of the B language, a part of the B method, not only as a specifications language but also as a modelling language for a wide range of challenging constraint based problems. We show on several puzzles and case studies that it is possible to formalize these problems elegantly using B. We also show that for many problems it is possible to solve these formalizations using ProB.
We use our results on a larger case study about performing data validation of university curricula using a formal specification. In this case study we use the B language to model and validate the feasibility of university curricula from a students' perspective and show that ProB can efficiently solve this validation problem. In particular, we show that it is possible to embed ProB in an application and solve this validation problem at runtime using our B model.
Afterwards, we will present a general structure of a data validation project in B and outline common challenges along with various solutions. This discussion is rooted in the results and experiences gathered on our case study and on a second independent one. We also discuss possible evolutions of the B language to make it (even) more suitable for such projects.

In the course of this thesis we discuss several alternative modelling approaches and how they relate to B and ProB. To conclude, we perform an in-depth evaluation where we compare our B and ProB based approach to several other tools and languages that can be used for this kind of validation task. We conclude that our approach of using B not only as a formal specification language but also as a constraint modelling language can be applied successfully in this scenario. Nevertheless, there are areas where this approach could be improved or extended to better suit this kind of application. We also conclude that ProB produces very good results for the high abstraction level of the language, that it is in many cases faster than brute force solutions and that it is comparable to dedicated constraint solving approaches.

Zusammenfassung

Formale Methoden bieten ausdrucksstarke Spezifikationssprachen um Probleme systematisch zu analysieren und diese mit einem hohen Abstraktionsgrad zu beschreiben. Mit einer Spezifikation werden Eigenschaften eines Systems beschrieben. Um deren Korrektheit zu überprüfen sind Softwarewerkzeuge zentral. Diese erlauben es durch unterschiedliche Techniken, wie Beweise oder Model-Checking, solche Spezifikationen zu verifizieren.

Ist es darüber hinaus möglich nicht-triviale Constraint-Satisfaction Probleme in diesen Spezifikationssprachen auszudrücken und ein solches Model zur Laufzeit zu verwenden um Probleme zu lösen und Datenvalidierung durchzuführen? Sind die existierenden Sprachen und Werkzeuge ausdrucksstark und mächtig genug für diesen Einsatzzweck? Dies sind die zentralen Fragen, die in dieser Arbeit erörtert werden. Sie werden mit einem besonderem Augenmerk auf die B-Methode – eine zustandsbasierte formale Methode zur Softwareentwicklung – und dem ProB Werkzeug – ein Animator und Model-Checker für die B-Methode – untersucht.

Zunächst wird die Verwendung der B-Sprache, ein Teil der B-Methode, nicht nur als Spezifikationssprache sondern auch als Sprache um Constraints zu beschreiben diskutiert. Anhand einer Reihe unterschiedlicher Puzzles und Fallstudien wird gezeigt, dass es möglich ist, diese mit der B-Sprache auf elegante Art und Weise zu formalisieren und dass es möglich ist, diese Problembeschreibungen mit Hilfe vom ProB auszuwerten.

Diese Ergebnisse bilden die Grundlage einer umfangreichen Fallstudie über die Validierung von Hochschulstudienplänen mit Hilfe einer formalen B-Spezifikation. Insbesondere wird gezeigt, dass es möglich ist ProB in eine Anwendung einzubetten um dieses Validierungsproblem zur Laufzeit zu lösen.

Aufbauend auf den Ergebnissen und Erfahrungen, die bei der zuvor erwähnten Fallstudie und bei einer zweiten unabhängigen Fallstudie gesammelt wurden, wird eine allgemeine Struktur für B-basierte Datenvalidierungsprojekte vorgestellt. Außerdem werden Probleme, die bei diesen Projekten auftreten können, sowie mögliche Lösungsansätze diskutiert.

Im Verlauf dieser Arbeit werden verschiedene alternative Modellierungsansätze diskutiert, sowie mit B und ProB verglichen. Den Abschluss dieser Arbeit bildet ein systematischer Vergleich dieses auf B und ProB basierenden Ansatzes zu alternativen Methoden, die für die untersuchten Validierungsprobleme eingesetzt werden können. Aus diesem Vergleich lässt sich schließen, dass B nicht nur als formale Spezifikationssprache sondern in diesem Kontext auch als Costraint-Modellierungssprache eingesetzt werden kann. Eines der Ergebnisse dieser Arbeit ist, dass ProB sehr gute Ergebnisse bei der untersuchten Art von Validierungsproblemen, insbesondere unter Berücksichtigung des Abstraktionsgrades der Sprache, liefert. Ferner ist ProB in vielen Fällen schneller als Brute-Force-Lösungen und liefert Ergebnisse, die mit dedizierten Constraint-Solving-Ansätzen vergleichbar sind.

Acknowledgments

During the time I have worked on this thesis I have received help, support and encouragement from many people, to whom I am deeply thankful.

Firstly I would like to thank my advisor Michael Leuschel for his guidance, support and the chance to explore different ideas, some of which finally lead to this thesis. I am also very grateful for the opportunity to work with him and be a part of his team.

I'd like to thank all off my current and former colleagues and students at the chair of software engineering and programming languages with whom I had the pleasure to work during my time as a student and later as a member of the research group. Additionally a honourable mention goes to our foosball table.

A central part of the work discussed in this thesis is the timetable validation case study discussed in Chapter 4 and Chapter 6. In this context I would like to particularly thank Frank Meier for initially bringing up this project and his continuous involvement in the project. Thanks also go to Svenja Wrede, Michael Heim, Nicolas Carsten, Nico Steffen, Niklas Bertram, Sarah Hönerbach and Jürgen Rauter from the deans' offices at the faculties of *Arts & Humanities* and *Business Administration & Economics* at Heinrich Heine University Düsseldorf for the very fruitful collaboration and their helpful feedback during the project. Particular thanks go to Tobias Witt, Philip Höfges and Joshua Schmidt who put very much of effort into the implementation of the timetable validation application and without their work it would have been impossible to create the application that was the result of this case study.

My deep gratitude also goes (alphabetically) to Anna Michaelis, Carl Friedrich Bolz, Dominik Hansen, Frank Meier, Michael Leuschel, Nicole Grothe and Stephan Zalewski for proof-reading and commenting on different draft versions of this thesis as well as for the inspiring and insightful discussions to which their feedback led. Regarding the articles that have been used in this thesis, I would like to thank the anonymous reviewers of ABZ 2014, FM 2015 and ABZ 2016 for their work and valuable feedback.

Finally, I would like to extend a special thank you to my family and friends, in particular to my wife Anna, for their patience, curiosity about my work and most importantly of all for their support.

Contents

I Introduction and Background **1**

1 Introduction **3**

2 Background **7**
2.1 Formal Methods & Specification Languages 7
2.1.1 The B Method 9
2.1.2 Formal Data Validation 11
2.2 Constraint Programming 13
2.3 ProB - Constraint Solving for the B Method 14

II B for Constraint Modelling and Data Validation **19**

3 Using B as a Constraint Modelling Language **21**
3.1 On the Expressiveness of B - The Jobs Puzzle 23
3.1.1 Shapiro's Challenge 24
3.1.2 A Solution to the Jobs Puzzle Using B 25
3.1.3 Related Work 29
3.2 Solving Constraint Problems with B and ProB 30
3.2.1 Subset Sum 31
3.2.2 n-Queens 33
3.2.3 Peaceable Armies of Queens 34
3.2.4 Extended Graph Isomorphism 37
3.3 Industrial Applications 38
3.4 Conclusion and Future Work 40

4 Case Study: Timetable Validation and Improvement **43**
4.1 Background 43
4.2 A Domain Theory of Timetables and Curricula 45
4.2.1 Background 45
4.2.2 Timetabling Data 46
4.2.3 Module Combinations 49
4.2.4 Curricula/Timetables 53
4.3 Modelling Curricula and Timetables in B 59
4.3.1 Data Import & Representation 60
4.3.2 Validation 69

4.3.3 Assisting the Solving Process 72
4.3.4 Computing Conflict Sources 73
4.3.5 Improvement – Finding Alternative Time Slots 75
4.3.6 Model Structure 78
4.3.7 Modelling Stages 78
4.4 From Model to Application 80
4.4.1 System Architecture 80
4.5 Related Work and Discussion 85
4.6 Future Work and Conclusion 87

III Evaluation and Outlook 89

5 Evaluation of the Software Solution 91
5.1 Introduction 91
5.2 Background 91
5.3 The Big Picture 93
5.4 Preparing Data for Use With a B Model 95
5.5 Data Representation 96
5.6 Means of Abstraction – Structuring and Auxiliary Constructs 98
5.6.1 Machines and Operations 98
5.6.2 Expressions and Predicates 99
5.6.3 DEFINITIONS 100
5.7 Using B to Express Computations 102
5.7.1 Machines and Operations 102
5.7.2 Recursive Functions 104
5.7.3 External Functions 105
5.7.4 Further Language Extensions 106
5.8 Interaction With the Model 108
5.9 Configuration Management 109
5.10 Conclusion and Future Work 109

6 Comparative Evaluation 111
6.1 Introduction 111
6.1.1 Scope 112
6.2 Data Model for Evaluation 112
6.2.1 Description of the Data Model 113
6.2.2 Validation 114
6.2.3 Data Sets 116
6.3 Languages and Tools 117

6.4 Evaluation 119
6.4.1 Prolog 120
6.4.2 SMT-LIB/Z3 135
6.4.3 Alloy 145
6.4.4 Python 151
6.4.5 ProB/B 161
6.5 Discussion 165
6.5.1 Enumeration 165
6.5.2 Unsatisfiable Core 167
6.5.3 Performance 168
6.5.4 Complexity 170
6.5.5 Conclusion 176

7 Conclusion and Future Work **179**

IV Appendices **183**

A Publications **185**
A.1 Towards B as a High-Level Constraint Modelling Language 185
A.2 Model-Based Problem Solving for University Timetable Validation and Improvement 186
A.3 Using B and ProB for Data Validation Projects 187

B Towards B as a High-Level Constraint Modelling Language **189**
B.1 Alloy *n-Queens* Model 189
B.2 Alloy *Peaceable Armies of Queens* Model 191

C B Machine to Validate Module Combinations for Different Courses **193**

D Evaluation **197**
D.1 Data Sets 197
D.2 Prolog 200
D.3 SMT-LIB/Z3 204
D.4 Python 207
D.5 ProB 211

Bibliography **223**

List of Figures **239**

List of Tables **243**

Part I

Introduction and Background

1 Introduction

Is it possible to express non-trivial constraint satisfaction problems in B and to use such a formal model at runtime for problem solving? Are the language and the tools powerful enough to enable this usage scenario? These are the central questions we explore in this thesis.

The B Method [3], a formal method for software development introduced in the last decade of the twentieth century by J.R. Abrial as a successor to Z [70, 120]. B is a formal method for specifying safety critical systems, reasoning about those systems and generating code following the *correct by construction* approach. The B specification language, part of this method, is a rich mathematical language based on an abstract machine notation and built around the concepts of first order logic, higher-order relations and set theory. These properties allow the users of the language to formalize and express complex problems in a succinct and elegant manner at a high level of abstraction.

The central matter of this thesis are applications of the B specification language using ProB [88] beyond formal systems validation. ProB is an automatic animation and model checking tool for B Method and other, mainly state based formal methods. The tool is developed and maintained at the chair for programming languages and software engineering (STUPS) at Heinrich Heine University Düsseldorf.

Due to the characteristics of B, ProB gradually evolved into a constraint solving tool for the B language, in order to automatically determine values for parameters and quantified variables in as many cases as possible. Guided by research and industrial usage, ProB's constraint solver for the B language has been continually improved in order to support more complex specifications. This has opened up new uses of B beyond developing safety critical systems, i.e. as a modelling language for constraint based problems.

An area where these features have successfully been applied is that of (formal) data validation [2]. In this area a formal model of a data set is used to validate, possibly very large, data sets that are used by a system at runtime. The formal data model captures the requirements on the data needed to ensure correct system behaviour. Data sets corresponding to different system instances can then be checked using ProB's constraint solver or other tools against the properties they must satisfy according to the formal model of the data. The data validation approach has successfully been used in areas where a formal model is created in a generic manner to avoid repeated development costs and configured at runtime, where the verification tools might not be able to handle large data sets [87] or where the validation of data previously was done manually [2].

In this thesis we want to look in more detail at how well suited parts of the B language are as a means to model constraint based problems. Linked to this question, we also want to explore how well suited ProB's constraint solver, a tool created for a different usage scenario, is for solving challenging constraint validation problems specified in (a subset of) B.

Assuming that ProB is able to handle complex constraint based validation problems, we ultimately want to explore if it is possible to build an application that uses ProB as part of its runtime. The goal would be to embed ProB and a B model in an application and interact with it as a tool or library at runtime, without having to generate code from our model.

All in all, the goals of this thesis are:

(i) Evaluate the use of B not only as a formal modelling language but also as a constraint modelling language.

(ii) Evaluate if ProB can be used to efficiently find solutions to complex constraint problems modelled in B.

(iii) Explore if ProB can be used as a runtime for constraint based models and embedded in applications.

(iv) Analyse how the combination of B and ProB compares to other tools that can be used to model constraint based problems.

Before trying to answer the questions above, we introduce in Chapter 2 the different concepts used in this thesis and provide some background on the context in which our work has taken place. In Chapter 3 we focus on the B language and explore its use to model constraint based problems and present a few simple benchmarks showing ProB's performance on these types of problems. Having explored the usability of B for constraint modelling, Chapter 4 concetrates on a larger case study, about the validation of university timetables. On this case study we evaluate if these claims can be applied to complex constraints in the context of a larger application. In this chapter we introduce and formalize a university timetable validation problem. Based on this problem we try to determine if it is feasible to take the B language from the comparatively simple constraint modelling problems in Chapter 3 to a larger problem with real world data and applications. Additionally, we explore if and how it is possible to create an application where our B model and ProB are not only used as independent validation tools, but are integral parts of the application, in the sense that our B model is running on top of ProB within the deployed application.

Chapter 5 revolves around the experience gathered while implementing the aforementioned case study and an independent data validation project. Based on these projects we try to outline a common structure for data validation projects. Additionally, we present different challenges encountered when using B for data validation, discuss different language constructs of B and argue how they can be applied in modelling data validation problems. Moreover, we outline areas where we have extended the B language to overcome some limitations we faced evaluating the models with ProB.

In the course of this thesis we discuss several alternative modelling approaches and how they relate to B and ProB, before doing a detailed evaluation of some of these approaches in Chapter 6. In this chapter we aim at comparing our use of B and ProB to different languages and tools that can also be used to model constraint based problems. The evaluation is based on a simplified version of the problem discussed in Chapter 4. We compare different tools and languages with regard to the complexity of modelling the problem and the performance of the associated tool to find a solution to problem.

Lastly, in Chapter 7 we recapitulate the work discussed in this thesis, presenting overall conclusions and providing an outlook into future work.

2 Background

In this chapter we will introduce the several concepts that build the basis for the work presented in this thesis. First we give a brief introduction to the concept of formal methods and formal specification languages with a particular focus on the B Method, we will present the idea of data validation and constraint satisfaction problems in general and present the ProB tool.

2.1 Formal Methods & Specification Languages

Formal methods are an approach to software and systems engineering that focuses on the validation and verifiability of software and hardware designs. This approach is used in particular in the context of safety critical systems, e.g. train control systems. Formal methods are used in areas where a high degree of confidence in the correct and safe execution of a hardware or software design is critical and a failure might endanger human lives or significantly damage the environment [77]. Particularly in the area of software development for railway systems the use of formal methods is common. This is due to the fact, that their use is *highly recommended* in the European standard EN50128 [29] for systems with a high safety integrity level [47].

Formal methods are also applied in mission, business and security critical contexts, where a system is essential to an organization [65, Table 2]. These applications of formal methods can range from their use to design systems that ensure data protection, avoid unauthorized access or ensure the correctness of business critical algorithms [104] among many other aspects. These are cases where failure might not have catastrophic consequences that put people at risk, but might cause financial or other form of damage to an organization [77].

Formal methods are built around formal specification languages, which provide the means to formalize and structure problem descriptions. These languages are generally based on mathematical rigorous notations and techniques that are used to “specify, model, develop and reason about computing systems” [118, p. 24] at a high level of abstraction. Following the definition provided by the NASA Langley Formal Methods Research Program, mathematical rigour means that formal methods and specification languages are based on mathematical logic. This makes the verification of a system specification a mathematical deduction process using the logic of the chosen formalism.[1] With a method that is well-defined and founded on mathematical logic it is possible to symbolically analyse a specification in order to validate properties covering all possible states or configurations described by the specification. Analysis can either be performed by conducting a proof using the rules of the logic or by systematically exploring all possible configurations (model checking) defined by the specification according to the rules of the chosen logic.

There is a large variety of languages, methodologies and tools in the domain of formal methods as well as many ways to apply these in the process of software and system modelling. Different methods and tools are suited for different use cases, to validate different aspects of a design or to be applied at different levels of abstraction, etc. Notable examples of formal methods are TLA$^+$ [83], CSP [66], VDM [20], Alloy [72] and the B Method [3] (see next section) among many others.

The mathematical rigour is also helpful to reason about a design at a higher abstraction level to better capture a problem and its details. Available verification tools can be used to explore the correctness of a design.

The approach of formal methods is used to manage the complexity in validating system designs and developing correct software. Formal methods provide languages and formalisms to support people modelling their systems and ensuring their correctness through a variety of techniques, such as correctness proofs, model checking etc. Most formal methods approaches and specification languages are associated with tools either created in conjunction with the formalism or created specifically to support the respective validation process. Some examples of tools for different formal methods are the proof system TLAPS [30] and the model checker TLC [131] for TLA$^+$, the Alloy Analyzer for

[1]`http://shemesh.larc.nasa.gov/fm/fm-what.html` - [Online; accessed 31-March-2017]

Alloy [72, p. 152], Atelier-B [31] and ProB [88] (discussed later in this chapter) for the B Method or the Spin model checker for the Promela language [15].

2.1.1 The B Method

In this thesis we will focus on aspects of a specific formal method, namely the B Method [3], and one of our goals is to explore its usability as a constraint modelling language.

The B Method is a formal method for software development introduced in the last decade of the twentieth century by J.R. Abrial as a successor to Z [70, 120], with the first scientific conference specifically about B taking place 1996 in Nantes, France [59]. An extension and simplification of the method, named Event-B [4], was later introduced, which focuses on system-level design and analysis.

The B Method is, based on Abrial's high-level characterization of B [3], a "method for specifying, designing and coding software systems" [3, p. xv] that covers all aspects of the software life cycle, of which he names specification, design through refinement, layered architecture and code generation. In the method, each of these named activities is backed by proofs in order to guarantee its correctness. This type methodology is often described as a correct-by-construction approach, where an implementation is derived systematically from a mathematical model of the system.

The specification language is built around models which are formalized using the Abstract Machine Notation (AMN) [3, p. 227]. This notation serves as a structuring, abstraction and encapsulation mechanism similar to classes or modules in programming languages.

The state of a machine is always private to it and can only be accessed and manipulated via operations. It is described using the mathematical concepts built into the language, such as sets, functions, relations, first order logic, etc. B is a state based method, the values of the data characterize the machine's state. In addition to the variables the states of a machine are characterized by an invariant property. This property is a predicate that describes the set of valid states the machine can be in by constraining the set of allowed values for each variable. Operations are used to describe state transitions. Operations consist of a precondition, describing when the operation can be executed, and of an atomic action that describes changes to the state variables. The precondition must be satisfied in the machine's current state in order for the operation to be executable. The action, defined using a concept of generalized substitution, describes how the successor state is computed from the values of the variables in the current state. Refinement, an important aspect of the method, is the process of stepwise transforming the model to executable code. In each refinement step the original abstract model is extended with concrete data types and operations while maintaining the interface described by the abstract model. The correctness of each refinement step is backed by a proof, to guarantee that original and refined machines are correctly associated. Through successive refinement it is possible to generate code, that, with regard to the abstract specification, is correct by construction.

A simple example of a B machine, in AMN, is shown in Figure 2.1. The machine is composed of several sections The `CONSTANTS` section introduces names for values that do not change once initialized. The domain of each constant is described in the `PROPERTIES` section in form of a predicate that constrains the range of possible values. The `VARIABLES` section introduces, as the name indicates, variables that are used in the machine. These variables are modified by the operations defined in the `OPERATIONS` section. Changes to these variables represent state transitions. In our example of a simple counter every time we increment or reset the counter we transition to a new state.

State transitions in the example are created by executing either the `Count` or `Reset` operations. The latter of the two operations can be executed from any state. Whereas the `Count` operation can only be executed from a state where the guard or precondition of the operation is true. Guards of preconditions are used to restrict possible transitions, i.e. to only transition into valid states; in this case we only allow the counter to be incremented while it hasn't reached its maximum value.

As discussed above, the invariant of a model describes a property that should hold in every possible state of a machine to ensure the integrity of the system. In our simple example we require the variable `count` to be a natural number and to be less than the value `Max` in every state the system can reach. Violating this indicates that the system has reached an undesirable state. Reaching such a state indicates that the requirements are wrong or there is an error in the modelling.

```
MACHINE Counter
  CONSTANTS Max
  PROPERTIES Max = 10
  VARIABLES count
  INVARIANT count : NATURAL & count < Max
  INITIALISATION count := 0
  OPERATIONS
    Count = PRE count < Max
      count := count + 1
    END;
    Reset = BEGIN
      count := 0
    END
END
```

Figure 2.1: B Machine modelling a counter that can be incremented until it reaches its predefined maximum value. This model contains an error.

In our example we can reach a state, whose configuration violates the invariant, namely the state where $count = Max$, which is a contradiction to the invariant that requires the variable `count` to always be less than `Max`. To solve this problem we can either loosen the invariant or strengthen the precondition to make it impossible to reach this state.

2.1.2 Formal Data Validation

As described so far, formal methods are a powerful method to verify and guarantee the correctness of a system or software. In particular with tool support it is possible to verify large projects using automatic and semi-automatic provers.

The formal correctness and correct behaviour of a system do not only depend on its implementation but also on the provided configuration and the data upon which a system acts. This is in particular true for systems that are modelled and validated in a generic manner and later configured with data for a specific instance or scenario, e.g. for a specific track layout in the case of a railway deployment [8, 87].

The generic way of modelling has the advantage, that the developed system has to be verified only once [9]. The behaviour of each instance of the system depends on the correctness of the data used, hence the correctness of said data is central to the well-behaved working of the system itself.

Nevertheless, when dealing with large data sets in the models many provers reach their limits as to what they can process [87, p. 431]. Additionally, for generic models, the correctness of the configuration data has to be ensured independently.

Generally, data validation can be understood as the process of ensuring that a set of data used in a safety-critical computer system satisfies rules and constraints expected by a consumer. Rules can express usefulness, completeness or correctness criteria on given data or on derived properties. In this sense, formal data validation is ensuring a data set satisfies a given formal model of the rules and relationships in the data [2, 85].

Originally the validation of data, even in a formal development process, was done manually [2]. This process could be very expensive, could take a long time and due to its manual nature is error prone. Lecomte et al. report in “Formally Checking Large Data Sets in the Railways” [84] that, as part of the development of the software for the metro line 14 in Paris, the process of manually validating 100000 items against 200 rules took more than six months.

The idea of formal data validation is to create a formal model of the data and the rules a data set has to satisfy. Such a specification explicitly captures the properties the data has to satisfy. This can go from checking simple properties, such as checking the types of values, to ensuring that certain indirect relations are present or that rules about aggregated properties of the data are satisfied. A formal model of data can be validated using either specialized tools or standard tools that can manage larger data sets. The formalized rules are evaluated on the full data set to ensure the data satisfies the requirements encoded in the rules, and if not to find counter examples.

For projects already using B for the software specification, the choice of B as the language to specify the rules the for the data is not far fetched. When using B for data validation, the data is described mainly using the mathematical language in B. The B Method has been used several times in industrial applications to perform data validation. ProB has been successfully used to perform data validation on such specifications [86].

Some approaches using B or Event-B as a data validation tool have used the language and tools to explicitly formalize a data model and validate it [86]. Other approaches, such as OVADO [2] or DTVT [84], are tools created on top of the B language that provide support for creating formal models of data. Both of these tools generate a formal B model of the data, evaluate them and provide feedback to the user. DTVT is based on ProB, whereas OVADO uses a redundant toolchain based on ProB and PredicateB [86] in order to verify the results provided by either tool. Of course there exist many other languages and tools that are very well suited for this task and which are used in similar domains, for instance the Datalog [52] based RailCOMPLETE tool [95].

The approach of tool based formal data validation to automatically validate configuration rules against specific instances can drastically reduce the validation time. For instance, as Leuschel et al. report in "Automated property verification for large scale B models with ProB" [90] in an industrial case study it was possible to reduce the validation time from about a month for manual inspection to several minutes using ProB.

2.2 Constraint Programming

Constraint Programming (CP) is a declarative programming paradigm based on describing a problem by means of the relation between its variables.

Following Apt's definition in "Principles of Constraint Programming" [6, p. 9] we define constraint and constraint satisfaction problems as follows:

Given a sequence of variables $Y := y_1, ..., y_k$ where $k > 0$, with domains $D_1, ..., D_k$ such that each variable y_i ranges over the domain D_i. Domains frequently range over boolean or numeric values but can be arbitrary sets of objects. A **constraint** C on Y is a subset of the Cartesian product $D_1 \times ... \times D_k$ of the domains. A **constraint satisfaction problem** (**CSP**) is a finite sequence of variables $X = x_1, ..., x_n$ with corresponding domains $D_1, ..., D_n$ together with a finite set $\mathcal{K}$ of constraints each on a subsequence of X.

A solution to a CSP is a sequence of legal values to all of its variables such that all its constraints are satisfied. An n-tuple $(d_1, ..., d_n) \in D_1 \times ... \times D_n$ **satisfies** a constraint $C \in \mathcal{K}$ on variables $x_{i_1}, ..., x_{i_m}$ if $(d_{i_1}, ..., d_{i_m}) \in C$. Such an n-tuple $(d_1, ..., d_n)$ is a

solution to a CSP if satisfies every constraint $C \in \mathcal{K}$, where $\mathcal{K}$ is the CSP's set of constraints. If the CSP has a solution it is **consistent**, else it is **inconsistent**.

Example Given a CSP over variables (a, b) with domains $a \in 1..4$, $b \in 1..4$ and constraints $a < b$ and $a + b < 5$. The constraint $a < b$ represents the set $\{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$ domain value pairs that satisfy the constraint. Whereas the constraint $a + b < 5$ represents the set of pairs $\{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (3, 1)\}$ of domain values. A solution to the CSP must be an element of $1..4 \times 1..4$ an satisfy both constraints of the CSP. Each of the tuples $(1, 2)$ and $(1, 3)$ represents a solution to the CSP.

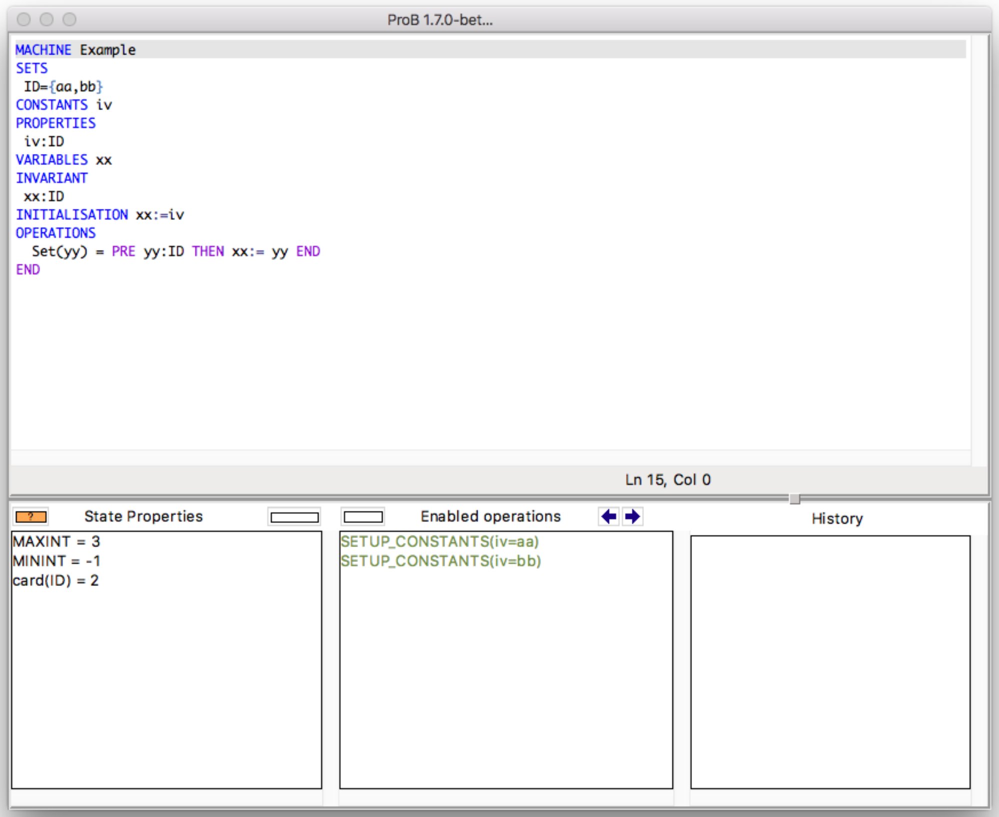
The process of formalizing a problem as a CSP is often referred to as **modelling**. The formalization consists of defining a set of decisions variables for the problem and associating them with constraints that must be satisfied [50]. The constraint programming process consists of modelling a problem as a CSP and solving it using either domain specific or general methods. Domain specific methods are algorithms developed for solving CSP in areas where custom techniques exist. Unification algorithms, custom solvers for problems based on integers or SAT and SMT solvers are examples of domain specific solving techniques. General techniques are techniques concerned approaches aimed at reducing the search space and searching for solutions within this space.

As stated by Apt one of the central goals in the research of constraint programming is to find and develop efficient techniques for solving CSP in different domains [6].

As there are many different approaches and tools there are also numerous areas where constraint programming is applied. Applications range from the layout of user interfaces by expressing geometric relations [10] to scheduling systems [12] and test data generation [56] among many others.

2.3 ProB - Constraint Solving for the B Method

The central tool for this thesis is the ProB animator and model checker [88]. ProB initially created for the B Method nowadays supports several state based formal methods, such as Event-B [17], Z [107, 108], TLA$^+$ [60], and also CSP [91]. ProB's central


ProB 1.7.0-bet...

```
MACHINE Example
SETS
 ID={aa,bb}
CONSTANTS iv
PROPERTIES
 iv:ID
VARIABLES xx
INVARIANT
 xx:ID
INITIALISATION xx:=iv
OPERATIONS
  Set(yy) = PRE yy:ID THEN xx:= yy END
END
```

Ln 15, Col 0

State Properties

```
MAXINT = 3
MININT = -1
card(ID) = 2
```

Enabled operations

```
SETUP_CONSTANTS(iv=aa)
SETUP_CONSTANTS(iv=bb)
```

History

Figure 2.2: Screenshot of ProB's user interface

features are the constraint solving and model finding capabilities built into the tool as well as its automatic animation features. By animation, in this context, we refer to the visualization and interactive exploration of the state space described by a model in order to validate properties of the formal model. Interactive exploration is achieved by evaluating transitions to other states, by means of operations in the case of a B model, and visualizing the results.

As Leuschel et al. describe in "From Animation to Data Validation: The ProB Constraint Solver 10 Years On" [87] the original motivation to create ProB in 1999 was the lack of **validation** tools among those available for the B Method. At the time the central tools for the B Method were BToolkit[2] and Atelier B[3], both of which provided mainly automatic and interactive proving environments and code generators.

In contrast to **verification**, i.e. ensuring the correctness of a model or specification, **validation** refers to the inspection of a model with the goal to ensure that it correctly captures the requirements and behaves as expected. BToolkit had restricted support for interactive animation that required the user to provide values for parameters and existentially quantified variables.

ProB's first goal was to provide automatic animation by computing values for parameters and existentially quantified variables, thus relieving the user from providing concrete values for these cases. To achieve this, ProB gradually evolved into a constraint solving tool for the B language. Since the language itself is undecidable, ProB cannot provide values for any conceivable B formula, but is able to efficiently solve a still growing subset of the language making it a viable and useful tool for validating and testing formal models.

Using the automatic animation features it is also possible to use ProB for model checking, i.e. systematically exploring the state space described by a model to verify properties and find states that violate certain conditions. E.g. finding a reachable state that violates the invariant or represents a dead-lock.

With the constraint solving features for the B language added to ProB, it can be used to ensure that large data sets satisfy properties described using a B model. This has led to the use of ProB as a data validation tool even in cases where B is not used for the system specification [2, 84, 87].

With ProB it is not only possible to check a large data set from a vast domain for its validity, but ProB can also be used to compute configurations under which a certain data set is valid with regard to given rules. As described in "Data Validation & Reverse Engineering"[4], the data validation/constraint solving features in ProB could be used in the process of reverse engineering an old IDE.

[2]`https://github.com/edwardcrichton/BToolkit` - [Online; accessed 31-March-2017]
[3]`http://www.atelierb.eu` - [Online; accessed 31-March-2017]
[4]`http://www.data-validation.fr/data-validation-reverse-engineering/` - [Online; accessed 31-March-2017]

In “Supporting Validation and Verification of State-Based Formal Models” [106] Plagge provides an overview of PROB’s architecture. At the core are the constraint solving and animation features which are implemented in SICStus Prolog using its clp(FD) implementation [27], that provides constraint solving over finite domains. PROB makes use of and extends clp(FD) in order to support the high-level features of the B language, such as infinite domains, quantified variables and arbitrarily nested sets, which has been described by Krings et. al. [80, 81].

Figure 2.2 shows a screenshot of PROB’s Tcl/Tk based user interface, consisting of an editor and three panes that show the current state of an animation and present controls to navigate the state space. Besides this user interface PROB can be invoked from the command line either in an interactive or batch mode. Additionally PROB can be used as a plugin for the Rodin Platform [17] and lastly a library has been created that exposes PROB’s features as a Java library [16].

Part II

B for Constraint Modelling and Data Validation

3 Using B as a Constraint Modelling Language

This chapter is based on a paper titled “Towards B as a High-Level Constraint Modelling Language” [93] presented at the ABZ 2014 conference.

In this chapter we argue that B is a useful language to conveniently express a wide range of constraint satisfaction problems. We also show that some problems can be solved quite efficiently by the ProB tool. We illustrate these claims on several examples, such as the *jobs puzzle* - for which we solve a challenge set out by Shapiro [116]. Here we show that the B formalization is both very close to the natural language specification and can still be solved efficiently by ProB. Our approach is particularly interesting when a high assurance of correctness is required. Indeed, compared to other existing approaches and tools, validation and double checking of solutions is available for ProB and formal proof can be applied to establish important properties or provide an unambiguous semantics to the problem specification.

The structure of this chapter is the following: first we present a case study which highlights the expressiveness of B as a constraint modelling language. The case study is based on the *jobs puzzle* [129] and takes on the challenges identified and discussed by Shapiro [116]. A part of this challenge is to provide a formalization of the puzzle that follows closely the English text of the puzzle. This aspect makes it particularly interesting, as it allows us to showcase how, using B, these kinds of problems can be expressed very conveniently and still be solved efficiently with ProB. The puzzle, the challenge and our solution to the puzzle are discussed and compared to other solutions in Section 3.1.

In a second step, we establish that ProB as a tool can be used to solve an interesting class of constraint satisfaction problems efficiently, providing a good balance between the ease of expressing a problem on an abstract level using B and the efficiency of solving these problems. In Section 3.2 we discuss a series of problems that are expressed nicely in B and can be solved by ProB for a wide range of values, such as the *n-Queens* problem, the *Peaceable Armies of Queens* and the *Graph Isomorphism* problem. We compare the results to a selection of different tools to show that ProB gives competitive results, while still having potential for improvement as discussed in Section 3.4. Finally, in Section 3.3 we present how the constraint solving features of ProB, showcased in this chapter, are being used in several industrial applications.

Alternative Approaches to Constraint Solving The mathematical language of B is quite close to that of Z and TLA$^+$. As ProB can deal with those formalisms [107, 60], the gist of the chapter is also valid for those languages. Similarly, VDM and abstract state machines are probably also well suited to express constraint satisfaction problems.

Dedicated Constraint Solving Libraries Alternate approaches to our high-level formal methods approach are dedicated constraint-solving libraries embedded in general purpose programming languages. Examples are the clp(FD) library of SICStus Prolog [27] or the IBM Decision Optimization Manager (formerly known as ILOG).[1] These libraries require a much higher modelling effort and a relatively high level of expertise, but can obviously obtain better performance. Another possible approach is the Zinc modelling language [97]. It provides a higher level encoding than for example clp(FD), but still cannot deal with higher-order sets or relations. Also, to our knowledge, neither Zinc nor any other tool we are aware of can deal with unbounded constraint satisfaction problems.

SMT-based approaches It would be interesting to see how an expert in the Formula language [73], which maps to the Z3 SMT solver, would encode the problems in this chapter, and how the solving times compare with those of ProB. Recently, an Event-B to SMT-LIB converter has become available for the Rodin platform [40]. It is very useful for proof, but as shown by Plagge and Leuschel [108] is not suitable for constraint solving. For example, it was not possible to solve various simpler problems, such as the *Who killed Agatha* puzzle or a graph colouring problem. As such, we did not attempt to use this B to SMT converter on the examples in this chapter. We revisit the use of SMT-LIB and Z3 for constraint based problems in Chapter 6.

[1] `http://www-03.ibm.com/software/products/en/decision-optimization-center` - [Online; accessed 14-March-2017]

SAT-based approaches The Alloy language [71] was designed from the outset to be able to effectively translated to SAT problems. This leads to certain expressivity restrictions, e.g., higher-order relations and sets are not allowed as their SAT encoding would become too large to be tractable. ProB follows another principle: it accepts the B language in full with all its consequences, and tries to solve constraints for as many relevant models as possible. Note that ProB also has a backend [108] which translates B constraints into SAT. This uses the same Kodkod library [125] that Alloy employs, and can deal much better with certain relational constraints, but similarly only translates first order sets and relations (the rest are left for the traditional ProB solver). In this chapter we discuss and compare several B solutions with Alloy counterparts and also discuss the ProB Kodkod backend.

Finally, one could think of using model checking rather than constraint solving. In fact, we have experimented with various solutions for the puzzles using efficient model checkers such as Spin or TLC. However, for constraint satisfaction problems model checking amounts to naive, brute force search and is rarely able to solve more complicated constraints.

3.1 On the Expressiveness of B - The Jobs Puzzle

The first and most detailed example we discuss is the *jobs puzzle*. This puzzle was originally published in 1984 by Wos et al. [129] as part of a collection of puzzles for automatic reasoners. A reference implementation of the puzzle, by one of the authors of the book, using OTTER [98] can be found online.[2]

The puzzle consists of eight statements that describe the problem domain and provide some constraints on the elements of the domain. The problem is about a set of people and a set of jobs. The question posed by the puzzle is: who holds which job? The text of the puzzle as presented by Shapiro in "The Jobs Puzzle: A Challenge for Logical Expressibility and Automated Reasoning" [116] is as follows:

- There are four people: Roberta, Thelma, Steve, and Pete.
- Among them, they hold eight different jobs.

[2]`http://www.mcs.anl.gov/~wos/mathproblems/jobs.txt` - [Online; accessed 14-March-2017]

- Each holds exactly two jobs.
- The jobs are: chef, guard, nurse, clerk, police officer (gender not implied), teacher, actor, and boxer.
- The job of nurse is held by a male.
- The husband of the chef is the clerk.
- Roberta is not a boxer.
- Pete has no education past the ninth grade.
- Roberta, the chef, and the police officer went golfing together.

What makes this puzzle interesting for automatic reasoners, is that not all the information required to solve the puzzle is provided explicitly in the text. The puzzle can only be solved if certain implicit assumptions about the world are taken into account, such as: the names in the puzzle denote gender or that some of the job names imply the gender of the person that holds it.

3.1.1 Shapiro's Challenge

Shapiro [116], following the original authors' remarks, that formalizing the puzzle was at times hard and tedious, identified three challenges posed by the puzzle with regard to automatic reasoners. According to Shapiro [116], the challenges posed by the *jobs puzzle* are to:

- formalize it in a non-difficult, non-tedious way.
- formalize it in a way that adheres closely to the English statement of the puzzle.
- have an automated general-purpose commonsense reasoner that can accept that formalization and solve the puzzle quickly.

Any formalization also needs to encode the implicit knowledge needed for the automatic reasoners to solve the puzzle while still trying to satisfy the aspects mentioned above. Addressing this challenge makes this puzzle a good case study for the expressiveness of B to formalize such a problem.

3.1.2 A Solution to the Jobs Puzzle Using B

The B encoding of the puzzle uses plain predicate logic, combined with set theory and arithmetic. We will show how this enables a very concise encoding of the problem, staying very close to the natural language requirements. Moreover, the puzzle can be quickly solved using the constraint solving capabilities of ProB. Following the order of the sentences in the puzzle we will discuss one or more possibilities to formalize them using B.

To express "*There are four people: Roberta, Thelma, Steve, and Pete*" we define a set of people, that holds the list of names:

```
PEOPLE = {"Roberta", "Thelma", "Steve", "Pete"}
```

We are using strings here to describe the elements of the set. This bears the advantage, that the elements of the set are implicitly different.[3] Alternatively, we could use enumerated or deferred sets defined in the `SETS` section of a B machine.

As stated above, in order to solve the puzzle we need some additional information not included in the puzzle's description. The first bit of information is that the names used in the puzzle imply the gender. In order to express this information we create two sets, `MALE` and `FEMALE` which are subsets of `PEOPLE` and contain the corresponding names.

```
FEMALE = {"Roberta", "Thelma"} & MALE = {"Steve", "Pete"}
```

The next statement of the puzzle is: "*Among them, they hold eight different jobs*". This can be formalized in B using a function that maps from a job to the corresponding person that holds this job using a total surjection from `JOBS` to `PEOPLE`:

```
HoldsJob : JOBS ↠ PEOPLE
```

Although redundant, as we will see below, to express "*Among them, they hold eight different jobs*" we can add the assertion that the cardinality of `HoldsJob` is 8. This is possible, because in B functions and relations can be treated as sets of pairs, where each pair consists of an element of the domain and the corresponding element from the range of the relation.

[3]This encoding allows us to input the puzzle directly into the ProB console.

```
card(HoldsJob) = 8
```

Constraining the jobs each person holds, the puzzle states: "*Each holds exactly two jobs*". To express this we use the inverse relation of `HoldsJob`, it maps a `PERSON` to the `JOBS` associated to her. The inverse function or relation is expressed in B using the `~` operator. For readability we assign the inverse of `HoldsJob` to a variable called `JobsOf`. `JobsOf` is in this case is a relation, because, as stated above, each person holds two jobs.

```
JobsOf = HoldsJob~
```

Because `JobsOf` is a relation and not a function, in order to read the values, we need to use B's relational image operator. This operator maps a subset of the domain to a subset of the range, instead of a single value. To read the jobs *Steve* holds, the relational image of `JobsOf` is used as shown below:

```
JobsOf[{"Steve"}]
```

Using the `JobsOf` relation we can express the third sentence of the puzzle using a universally quantified expression over the set `PEOPLE`. The universal quantification operator (∀) is expressed in B using the `!` symbol followed by the name of the variable that is quantified. This way of expressing the constraint is close to the original text of the puzzle, saying that the set of jobs each person holds has a cardinality of two.

```
!(x).(x : PEOPLE ⇒ card(JobsOf[{x}]) = 2)
```

The fourth sentence assigns the set of job names to the identifier `JOBS`. This statement also constraints the cardinality of `HoldsJob` to 8.

```
JOBS = {"chef", "guard", "nurse", "clerk",
        "police", "teacher", "actor", "boxer"}
```

The following statements further constrain the solution. First "*The job of nurse is held by a male*", which we can express using the `HoldsJob` function and the set `MALE` by stating that the element of `PEOPLE` that `HoldsJob`("*nurse*") points to is also an element of the set `MALE`.

```
HoldsJob("nurse") : MALE
```

Additionally, we add the next bit of implicit information, which is that typically a distinction is made between actress and actor, and therefore the job name actor implies that it is held by a male. This information is formalized, similarly as above.

```
HoldsJob("actor") : MALE
```

The next sentence: "*The husband of the chef is the clerk*" contains two relevant bits of information, based on another implicit assumption, which is that, at the time, marriage was only possible between one female and one male. With this in mind, we know that the chef is female and the clerk is male. One possibility is to do the inference step manually and encode this as:

```
HoldsJob("chef") : FEMALE & HoldsJob("clerk") : MALE
```

Alternatively, and in order to stay closer to the text of the puzzle we can add a function `Husband` that maps from the set `FEMALE` to the set `MALE` as a partial injection. We use a partial function, because we do not assume that all elements of `FEMALE` map to an element of `MALE`.

```
Husband : FEMALE ⤔ MALE
```

To add the constraint using this function we state that the tuple of the person that holds the job as chef and the person that holds the job as clerk are an element of this function when treated as a set of tuples.

```
(HoldsJob("chef"), HoldsJob("clerk")) : Husband
```

The next piece of information is that "*Roberta is not a boxer*". Using the `JobsOf` relation we can express this close to the original sentence, by stating: boxer is not one of Roberta's jobs. This can be expressed using the relational image of the `JobsOf` relation:

```
"boxer" /: JobsOf[{"Roberta"}]
```

The next sentence provides the information that "*Pete has no education past the ninth grade*". This again needs some context information to be useful in order to find a solution for the puzzle [116]. To interpret this sentence we need to know that the jobs of police officer, teacher and nurse require an education of more than 9 years. Hence the information we get is that Pete does not hold any of these jobs. Doing this inference step we could, as above, state something along the lines of `HoldsJob`(“*police*”) $\neq$ “*Pete*”, etc. for each of the jobs. The solution used here, tries to avoid doing the manual inference step. Although we still need to provide the information needed to draw the conclusion that Pete does not hold any of these three jobs. We create a set of those jobs that need higher education:

```
QualifiedJobs = {"police", "teacher", "nurse"}
```

Using the relational image operator we can now say that Pete is not among the ones that hold any of these jobs. The relational image can be used to get the set of items in the range of function or relation for all elements of a subset of the domain.

```
"Pete" /: HoldsJob[QualifiedJobs]
```

Finally, the last piece of information is that “*Roberta, the chef, and the police officer went golfing together*”, from this we can infer that Roberta, the chef, and the police officer are all different persons. We write this in B stating that the set of Roberta, the person that holds the job as chef, and the person that is the police officer has cardinality 3, using a variable for the set for readability.

```
Golfers = {"Roberta", HoldsJob("chef"), HoldsJob("police")}
&
card(Golfers) = 3
```

By building the conjunction of all these statements, ProB searches for a valid assignment to the variables introduced that satisfies all constraints, generating a valid solution that answers the question posed by the puzzle “*Who holds which job?*” in form of the `HoldsJob` function. The solution found by ProB is depicted in Fig. 3.1.[4]

This satisfies, in our eyes, the challenges identified by Shapiro. In the sense that the formalization, is not difficult, although it uses a formal language. The elements of this language are familiar to most programmers or mathematicians and it builds upon well understood and widely known concepts. The brevity of the solution shows that using an expressive high-level language it is possible to encode the puzzle without having tedious tasks in order to be able to solve the puzzle at all.

[4]We used the “Visualize State as Graph” command and then adapted the generated graph using OmniGraffle.

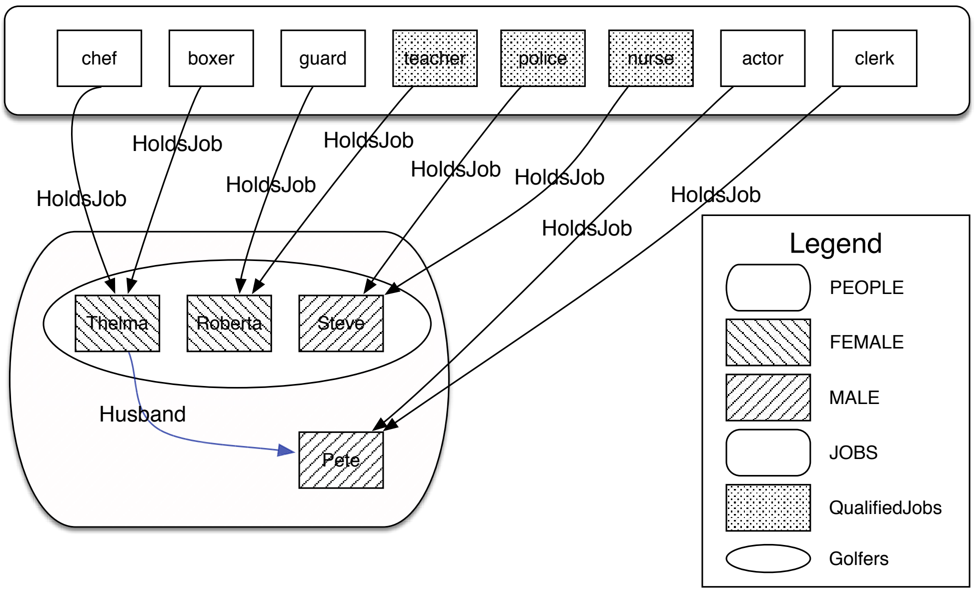


Figure 3.1: The solution to the Jobs puzzle, depicted graphically

The encoding of the sentences follows the structure of the English statements very closely. We avoid the use of quantification wherever possible and use set based expressions that relate closely to the puzzle. We are able to encode the additional knowledge needed to solve the puzzle in a straight forward way, that is also close to how this would be expressed as statements in English. Lastly it is worth to note that the formalization of "*Each holds exactly two jobs*" is the one furthest away from the English expression, using quantifications and set cardinality expressions.

3.1.3 Related Work

In his paper Shapiro discusses several formalizations of the puzzle with regard to the identified challenges. A further formalization using controlled natural language and answer set programming (ASP) was presented in "The jobs puzzle: Taking on the challenge via controlled natural language processing" [114] by Schwitter et al.

The first of the solutions discussed by Shapiro is a solution from the TPTP website, encoded as a set of clauses and translated to FOL. The main disadvantages of this encoding is that it requires 64 clauses to encode the problem and many of them are needed to define equality among jobs and names. In contrast to this, our B encoding uses either enumerated sets or strings, where all elements are implicitly assumed to be different. Thus the user does not have to define the concept of equality for atoms.

The second solution discussed by Shapiro uses SNePS [117], a common sense and natural language reasoning system designed with the goal to "have a formal logical language that captured the expressibility of the English language" [116]. The language has a unique name assumption and set arguments making the encoding simpler and less tedious. On the other hand the lack of support for modus tolens requires rewriting some of the statements in order to solve the puzzle.

The last formalization discussed by Shapiro uses Lparse and Smodels [105] which uses stable model semantics with an extended logic programming syntax. According to Shapiro several features of Lparse/Smodels are similar to those of SNePS. This formalization also simplifies the encoding of the puzzle, but according to Schwitter et al. both solutions still present a "considerable conceptual gap between the formal notations and the English statements of the puzzle" [114].

Schwitter et al. present a solution to the *jobs puzzle* using controlled natural language and a translation to ASP to solve the *jobs puzzle* in a novel way that stays very close to the English statements of the puzzle and satisfying the challenges posed by Shapiro. To avoid the mismatch between natural and controlled natural languages Schwitter et al. describe the use of a development environment that supports the user to input valid statements according to the rules of the controlled language. A solution using a mathematical, but high level language like B avoids this problems by having a formal and, for most, familiar language used to formalize the problem.

3.2 Solving Constraint Problems with B and ProB

ProB is able to solve our formalization of the *jobs puzzle*, presented in the previous Section, in about $10ms$, finding the two possible instantiations of the variables that represent the (same) solution to the puzzle.

In this section we will present four examples, that can be elegantly expressed with B and discuss how these can be solved with the constraint solving features of ProB.

3.2.1 Subset Sum

The first example is the subset sum problem 7.8.1 from page 340 of "Optimization Models For Decision Making: Volume 1".[5]

> A bank van had several bags of coins, each containing either 16, 17, 23, 24, 39, or 40 coins. While the van was parked on the street, thieves stole some bags. A total of 100 coins were lost. It is required to find how many bags were stolen.

Expressing this problem just takes this simple B snippet that ProB can solve it in less than 5 ms. The goal is to determine how many bags of coins were lost to amount for 100 missing coins. The bags can have different sizes, specified in the set named *coins*.

```
coins = {16,17,23,24,39,40} &
stolen : coins → NATURAL &
SIGMA(x).(x : coins | stolen(x) * x) = 100
```

In order to find the result with B, we create a function that maps from the different coin-bag sizes to a number; this number represents how many bags of that size were stolen. The instantiation of the function *stolen* is constrained by the last expression, which states that the sum of all coins in the missing bags is 100. This is expressed in B using the `SIGMA` operator, which returns the sum of all values calculated in the associated expression, akin the mathematical Σ operator. An interesting aspect is that we have not explicitly expressed an upper bound on *coins*. `NATURAL` stands for the set of mathematical natural numbers. ProB determines the upper bound itself during the constraint solving process. Finally, we can check that there is only one solution by checking:

```
1 = card({coins, stolen | coins = {16,17,23,24,39,40} &
                          stolen : coins → NATURAL &
                          SIGMA(x).(x : coins | stolen(x) * x) = 100})
```

[5] `http://ioe.engin.umich.edu/people/fac/books/murty/opti_model/` - [Online; accessed 13-March-2017]

Table 3.1: Time in *ms*. to find a first solution to the *n-Queens* problem.

Board Size n	ProB	C	Prolog	Prolog clp(FD)	Alloy MiniSat	Alloy Plingeling
8	163 ±10.05	3.25 ±0.48	0 ±0.0	10 ±0.0	181.9 ±14.81	1002.8 ±81.45
10	163 ±9.0	2.02 ±0.17	2 ±4.0	9 ±3.0	511.8 ±15.45	1634.2 ±161.08
20	181 ±10.44	42.13 ±2.65	194 ±4.90	10 ±4.47	11639.7 ±201.94	8158.9 ±2667.66
30	215 ±12.85	14204.94 ±30.52	107300 ±547.56	13 ±4.58	- -	33164.5 ±2414.33
40	246 ±4.90	- -	-* -	12 ±4.0	- -	- -
50	370 ±25.30	- -	- -	19 ±3.0	- -	- -
60	463 ±26.10	- -	- -	17 ±4.58	- -	- -
70	501 ±30.15	- -	- -	11 ±3.0	- -	- -
80	598 ±30.92	- -	- -	15 ±6.71	- -	- -
90	858 ±51.34	- -	- -	21 ±3.0	- -	- -
100	971 ±45.27	- -	- -	18 ±4.0	- -	- -

* We cancelled this run after 40 minutes without result.

3.2.2 n-Queens

The well known *n-Queens* problem[6] is a further problem, that can be expressed very succinctly in B by specifying a constant *queens*, which has to satisfy the following axioms:

```
queens : perm(1..n) &
!(q1, q2).(q1 : 1..n & q2 : 2..n & q2 > q1
    ⇒ queens(q1) + (q2 - q1) /= queens(q2) &
        queens(q1) + (q1 - q2) /= queens(q2))
```

The total and injective function *queens* maps the index of each column to the index of the row where the queen is placed in that column. The formula states that for each pair of columns, the queens placed on those columns are not on the same diagonal. From the set of functions from $1 \ldots n$ to $1 \ldots n$ ProB discards those candidates that violate the condition on the diagonals and instantiates *queens* to the first solution that satisfies it.

To get a better idea of how ProB performs for this, and hopefully similar problems we compared, as shown in Table 3.1, the B implementation on ProB 1.6.2-beta1 (revision: eec70f07) to a C iterative implementation, a version in Prolog using clp(FD) taken from the "SICStus Prolog user's manual" [28, p. 487], a version in Prolog taken from Chapter 14 of "The Art of Prolog" [121, p. 255] both running on SICStus Prolog 4.3.5, a version written in Alloy (See Appendix B.1) using the MiniSat [46] and Plingeling [18] SAT-solvers provided with Alloy 4.2_2015-02-22. We ran the examples on an otherwise idle machine running Linux Mint 18, with 4 GB of RAM and a quad-core Intel Core i5 CPU running at 2.67 GHz for increasing values of n. For each value of n we ran ten iterations of the example, each in a new process, and report the average runtimes in Table 3.1 All reported times represent the time needed to find a first valid configuration. For each tool we stopped collecting data after the first computation that took more than 10 seconds to find a solution.

There are some pathological cases, not shown here, where ProB, but also other tools perform very badly (such as $n = 88$) but for most inputs of $n < 101$ ProB finds a solution in less than a second. ProB has a noticeable overhead of about 160 milliseconds when evaluating the n-Queens predicate in a new process for the first time. Likely the

[6]`https://en.wikipedia.org/w/index.php?oldid=765435416` - [Online; accessed 13-March-2017]

overhead is the time the JIT compiler, added to SICStus Prolog in release 4.3, spends compiling some parts of ProB's core when we first evaluate an expression. This overhead is not present if the JIT is disabled via the `SP_JIT` environment variable. Also, it only affects the first execution of a predicate, subsequent executions of the same predicate are significantly faster.

The results show that constraint based solutions to this problem, with increasing board sizes, give better results than the brute force versions. Among the constraint based results, the direct encoding in Prolog using clp(FD) is generally faster than ProB; considering the higher abstraction level of B these results are to be expected. Taking the size of the implementations into account (as reported in Table 3.2) gives evidence that using B to encode such a problem and ProB to solve it is a good trade-off between the size, the complexity of the implementation and the time required to find a solution.[7]

Table 3.2: Implementation size in bytes for the different solutions compared

Language	Bytes
B (ProB)	141
Alloy	456
Prolog clp(FD)	1252
Prolog	1275
C	3059

3.2.3 Peaceable Armies of Queens

A challenging constraint satisfaction problem, related to the previous, was proposed by Bosch [24] and taken up by Smith et al. [119]. It consists of setting up opposing armies of queens of the same size on a $n \times n$ chessboard so that no queen attacks a queen of the opposing colour.

Smith et al. report that the integer linear programming tool CPLEX took 4 hours to find an optimal solution for $n = 8$ (which is 9 black and 9 white queens). Optimal here means placing as many queens as possible, i.e., there is a solution for 9 queens but

[7]We are aware that the different solutions compared might not represent the best possible solution in each formalism.

none for 10 queens. In order to determine the optimal solution for $n = 8$ the ECLiPSe and the ILOG solver are reported to take just over 58 minutes and 27 minutes and 40 seconds respectively (Table 1 in "Models and Symmetry Breaking for 'Peaceable Armies of Queens"' [119]). After applying the new symmetry reduction techniques proposed by Smith et al. [119], the solving time was reduced to 10 minutes and 40 seconds for ECLiPSe and 5 minutes 31 seconds for the ILOG solver.

In a first instance, we have encoded the puzzle as a constraint satisfaction problem, i.e., determining whether for a given board size n and a given number of queens q we can find a correct placement of the queens. We first introduce the following `DEFINITION`:

```
ORDERED(c,r) == (!i.(i : 1..(n-1) ⇒ c(i) <= c(i+1)) &
                 !i.(i : 1..(n-1)
                   ⇒ (c(i) = c(i+1) ⇒ r(i) < r(i+1))))
```

The encoding of the problem is now relatively straightforward:

```
blackc : 1..n → 1..dim & whitec : 1..n → 1..dim &
blackr : 1..n → 1..dim & whiter : 1..n → 1..dim &
ORDERED(blackc,blackr) & ORDERED(whitec,whiter) &

!(i,j).(i:1..n & j:1..n ⇒ blackc(i) /= whitec(j)) &
!(i,j).(i:1..n & j:1..n ⇒ blackr(i) /= whiter(j)) &
!(i,j).(i:1..n & j:1..n
    ⇒ blackr(i) /= whiter(j) + (blackc(i)-whitec(j))) &
!(i,j).(i:1..n & j:1..n
    ⇒ blackr(i) /= whiter(j) - (blackc(i)-whitec(j))) &

whitec(1) < blackc(1) /* symmetry breaking */
```

We have also encoded the problem in Alloy (See Appendix B.2). Table 3.3 shows the results for board sizes of 7 and 8 for ProB and Alloy. We have used Alloy in version 4.2_2015-02-22 and version 1.6.2-beta1 (revision: eec70f07) of ProB. We ran the examples on an otherwise idle machine running Linux Mint 18, with 4 GB of RAM and a quad-core Intel Core i5 CPU running at 2.67 GHz. We ran 10 iterations for each

Table 3.3: Runtime in milliseconds to solve the *Peaceable Queens* problem

Board Size	Queens	SAT	ProB		Alloy			
n	q				MiniSat		Plingeling	
7	7	✓	1614	±76.0	11837.2	±288.3	4672	±552.4
7	8	✗	23098	±189.6	-[*]	-	-[*]	-
8	1	✓	165	±12.7	59.7	±21.6	139.9	±4.3
8	2	✓	205	±9.7	102.1	±26.1	383.5	±83.9
8	3	✓	212	±21.0	300.8	±88.6	664.4	±127.4
8	4	✓	214	±10.8	760.2	±120.5	865.1	±19.7
8	5	✓	217	±16.4	1622.6	±292.7	1496.4	±266.6
8	6	✓	221	±14.5	2171.8	±168.9	1396.3	±240.5
8	7	✓	425	±7.1	6021.6	±631.0	4134.8	±434.0
8	8	✓	900	±45.2	7556.0	±599.5	5375.6	±428.3
8	9	✓	198501	±1036.7	21041.5	±881.3	11882.4	±3131.9
8	10	✗	1329552	±4409.8	-[*]	-	-[*]	-

[*] We cancelled this run after 30 minutes without result.

problem instance shown in the table and report the average runtime.

For small numbers of queens both SAT solvers we used with Alloy perform better than ProB. For feasible boards with up to 8 queens ProB gives better results than both SAT solvers, with a runtime of 900 milliseconds for 8 queens on a board of size 8 where Plingeling's solving time was 5375.6 milliseconds on average. For the optimal solution of placing 9 queens on a board of size 8 the results are different. In this case ProB does not perform as well as it did on the previous instances, here ProB takes 198501 milliseconds (3 minutes and 18 seconds) to find a solution. In comparison to this, Plingeling, the faster of the both SAT solvers on this instance, only takes 11882.4 milliseconds to find a solution. When checking the first unsatisfiable instance for each board size, i.e. placing 8 queens on a board of size 7 and 10 queens on a board of size 8, ProB is able to detect that these instances are not satisfiable in 23098 milliseconds and 1329552 milliseconds (22 minutes and 10 seconds) respectively. Alloy's SAT solvers had not detected the inconsistencies after 30 minutes when we cancelled each run. ProB can also solve the puzzle for $n = 8$, $q = 9$ and an additional two kings (one of each colour [55]). The solving time is about one hour.

This example has shown that even problems considered challenging by the constraint programming community can be solved, and that they can be expressed with very little effort. The graphical visualization features of ProB were easy to setup (basically just defining an animation function in B and declaring a few images; see "Easy Graphical Animation and Formula Viewing for Teaching B" [92] for details) and were extremely helpful in debugging the model.

3.2.4 Extended Graph Isomorphism

The *Graph Isomorphism* Problem[8] is the final problem discussed here. To determine if two graphs are isomorphic we search for a bijection of the vertices which preserves the neighbourhood relationship.

Using B, graphs can be represented as relations of nodes, where the vertices are represented by the tuples in the relation, seen as a set. An undirected graph can hence be easily represented as the union of the directed graph relation with the inverse of the relation, basically duplicating all vertices.

Using B we can state the problem, as shown in Figure 3.2, using an existential quantification over the formula used to define the *graph isomorphism* problem, following closely the mathematical problem definition. Existential quantification is expressed in B using the `#` operator, which corresponds to the $\exists$ symbol in mathematical notation. We state that there is a total bijection that maps the vertex set from one graph to the other such that two nodes are adjacent in the domain iff they are adjacent in the range of the bijection. Additionally we only need to encode the entities needed in the quantification in order to solve this problem for two specific graphs.

The specification in Figure 3.2 can easily be extended with additional constraints. An industrial application of such an extended *graph isomorphism* problem was presented by ClearSy.[9] Here, ClearSy used B and ProB to find graph isomorphisms between high-level control flow graphs and control flow graphs extracted from machine code gathered through a black box compiler. In addition, the memory mapping used by the

[8]`https://en.wikipedia.org/w/index.php?oldid=760657015` - [Online; accessed 14-March-2017]

[9]`http://www.data-validation.fr/data-validation-reverse-engineering/` - [Online; accessed 30-March-2017]

```
MACHINE CheckGraphIsomorphism
SETS Nodes = {a,b,c,d,e, x,y,z,v,u}
DEFINITIONS
  G1 == {a↦b, a↦c, a↦d, b↦c, b↦d, c↦e, d↦e};
  G2 == {x↦v, x↦u, x↦z, y↦v, y↦u, z↦v, z↦u}
CONSTANTS graph1, graph2, relevant
PROPERTIES
  graph1: Nodes <-> Nodes & graph2: Nodes <-> Nodes &
  /* generate undirected graphs */
  graph1 = G1\/G1~ & graph2 = G2 \/ G2~ &
  relevant = dom(graph1) \/ dom(graph2) \/ ran(graph1) \/
      ran(graph2) &
  #p.(p : relevant ⤖ relevant &
    !(x, y).(x : relevant & y : relevant ⇒
      (x↦y : graph1 ⟺ p(x)↦p(y) : graph2)))
END
```

Figure 3.2: B specification to check if two graphs are isomorphic.

compiler had to be inferred by constraint solving. For the main problem on graphs with 192 nodes each, the solution was found by ProB in 10 seconds. The ability to easily express graph isomorphism and pair it with other domain specific predicates was an important aspect in this application.

3.3 Industrial Applications

As mentioned in Chapter 2 and in the previous section, the expressivity of B in combination with the constraint solving capabilities of ProB are being used in several industrial applications in order to validate inputs for models or to solve problems similar to those shown in this chapter.

A further application by Siemens is described described by Leuschel et al. in “Automated property verification for large scale B models with ProB” [90]. Siemens uses the B Method to develop safety critical software for various driverless train systems. These systems make use of a generic software component, which is then instantiated for a particular line by setting a larger number of parameters (i.e., the topology, the style of trains with their parameters). In order to establish the correctness of the generic software, a large number of properties about these parameters and the system in general are expressed in B.

The data validation problem is then to validate these properties for particular data values. One difficulty is the size of the parameters; the other is the fact that some properties are expressed in terms of abstract values and not in terms of concrete parameter values (which are linked to abstract values via a gluing invariant in the B refinement process). Initially, the data validation was carried out in Atelier-B using custom proof rules [22]. However, many properties could not be checked in this way (due to excessive runtime or memory consumption) and had to be checked manually. Leuschel et al [90] describe that Siemens now uses the ProB constraint solver to this end and has thus dramatically reduced the time to validate a new parameterisation of their generic software. This success led to this technique also being applied in Alstom and the development of a custom tool DTVT with the company ClearSy [84]. The company Systerel is also using B for data validation for a variety of customers [9]. To this end, Systerel uses a B evaluation engine which is also at the heart of Brama [115] combined with ProB.

It is interesting to note, that data validation is now also being applied in contexts where the system itself was not developed using B; the B language is "just" used to clearly stipulate important safety properties or regulations about data. This shows a shift from using B to formally prove the correctness of systems or software, to using B as an expressive specification language for various constraint satisfaction problems. In the traditional use of B to develop software or systems correct by construction, refinement and the proving process play a central role. In this novel use of B, those aspects of B are almost completely absent. There often is no use of refinement but the properties to be checked or solved become larger and more difficult, making the use of traditional provers nigh impossible.

The ProB tool is now used by various companies for similar data validation tasks, sometimes even in contexts where B itself is not used for the system development process. In those cases, the underlying language of B turns out to be very expressive and efficient to cleanly encode a large class of data properties.

3.4 Conclusion and Future Work

We have discussed two aspects in this chapter. In the first part of this chapter we focused on the B language and its power to express constraint satisfaction problems. Using B, we have taken on the challenges identified by Shapiro regarding the *jobs puzzle*. Our primary goal was to show that while B is a formal specification language, it is also very expressive and can be used to encode constraint satisfaction problems in a readable and concise way. Our solution to the *jobs puzzle* addresses all the challenges posed by this puzzle, we were able to create an encoding, that using mainly simple B constructs, creates a simple and straight-forward formalization. We only have to additionally provide an encoding of the implicit information required to solve the puzzle, which can also be achieved in a way that is not complex and close to a translation to English. Our encoding follows the original text closely and ProB is able to solve the puzzle efficiently.

In the second part of this chapter we focused on solving constraint satisfaction problems written in B using ProB. We outlined on two similar examples that it is possible to efficiently solve problems encoded on a high abstraction level.

The pairing of ProB and B can be a good trade-off between the efficiency of low level constraint solvers that make the encoding of problems very hard and high-level systems that have to pay the price of abstraction by the increased amount of computation needed to solve problems. Surprisingly, for some problems (see Section 3.2.3) we are competitive with low-level modern constraint solving techniques. But obviously, many large scale industrial optimization problems are still out of reach of our approach. Also, compared to Alloy, the standard ProB solver is weak for certain relational operators such as image or transitive closure. But we work on further improving the constraint solving capabilities of ProB and thus reducing the overhead associated with the abstraction level of B, allowing to use ProB on more problems and domains.

As shown in the *Peaceable Queens* example, B and ProB are still awkward for solving optimization problems. The current solution is to setup a problem twice and search for a solution where one problem is solved and the other one not. This is an area we intend to do further research in the future.

Finally we discussed the industrial uses cases of B in combination with the constraint solving features of ProB. All the aspects discussed in this chapter show the advantages of using a feature rich and high-level language such as B to encode complex problems and at the same time making use of a high-level constraint solver to solve them. Compared to other approaches to constraint solving, ours has the advantage of extensive validation of the tool along with a double chain [84] to cross-check results, and the ability to apply proof to (parts of) B models. This makes B and ProB particularly appealing to solving constraints in safety critical application areas.

4 Case Study: Timetable Validation and Improvement

This chapter is an extended version of the article "Model-Based Problem Solving for University Timetable Validation and Improvement" [113].

As we have stated in the previous chapter, constraint satisfaction problems can be expressed very elegantly in state-based formal methods such as B. But can such specifications be directly used for solving real-life problems? In other words, can a formal model be more than a design artefact but also be used at runtime for inference and problem solving? We will try and answer this question in this chapter with regard to the university timetabling problem.

In this chapter we formalize a variant of the timetabling problem based on an ongoing project at Heinrich Heine University Düsseldorf (HHU). The goal of the project is to create a model based application to validate the feasibility of the offered curricula from a student's perspective We describe the problem domain, the formalization in B and our approach to execute the formal model in a production system using ProB.

4.1 Background

Motivation The main question pursued in this chapter is whether one can use this model-based approach to constraint solving already in a production system, or whether further research and development is required. In other words, is it possible to completely express a non-trivial constraint satisfaction problem in B (or some other state-based formal method), and to use this formal model in a real production system at runtime for inference and problem solving? The benefits would be substantial: expressing the constraints (correctly) would be considerably less difficult and modifying the constraints could be done declaratively within the formal model, with all the aid provided by formal methods and their tools. However, to be successful, the constraint solving capabilities need to scale to the real-life problem, they need to be robust and predictable, and one needs to be able to link the model with the graphical user interface in particular and the computing architecture in general.

The — maybe surprising — answer to this important question turns out to be positive. In this chapter, we show how we have successfully expressed a challenging timetabling problem at our university in B, and have developed a system which executes this formal model in real-time using the ProB constraint solver and provides a web-based graphical interface using a new API. The tools can be used to detect minimal conflict sets efficiently, providing the user with valuable feedback. Data is automatically imported from external sources and high-level constraints can be added or modified simply by editing the formal model.

University Timetabling Problem In cooperation with the faculty of *Arts & Humanities* (**AH**) and the faculty of *Business Administration & Economics* (**BAE**) at HHU we are working on a timetabling application to validate the feasibility of the offered curricula. For political and organisational reasons the decision was made to not generate completely new curricula based on the constraints we will describe below. Instead, the central goal of the project is to validate and improve timetables of all programs to ensure that it is possible for students to attend all classes required to finish their studies in the legal standard time, as defined by their chosen curriculum.

For this general feasibility we do not take into account aspects such as group sizes or room scheduling. Secondary goals of the project are to provide assistance to resolve conflicts in the timetables by computing conflict sets and feasible alternative time slots. Validating the provided timetabling data requires detecting scheduling conflicts, which would require a student, at any point in their studies, to attend two sessions at the same time. The complexity arises from the number of available combinations that need to be validated and the number of possible choices for students: some units are available in more than one semester and students can choose when to attend them. Units might be divided in sections or groups, offered on different days and times, where students have to choose one group and attend all sessions that are part of that group. Programs are often combinations of subjects and thus many programs share teaching units. This needs to be considered when searching for alternative time slots and modifying timetables.

Figure 4.1 shows a screenshot of the user interface of our application. The architecture of the application is divided into four components: we have a browser based user interface for interacting with the timetables, a server component built using Groovy that embeds the models using the ProB Java API [16][1] and exposes them to the front end using a REST (representational state transfer) based HTTP API, a storage interface that contains the timetable data and finally the model layer, which is built using classical B and evaluated using the ProB.

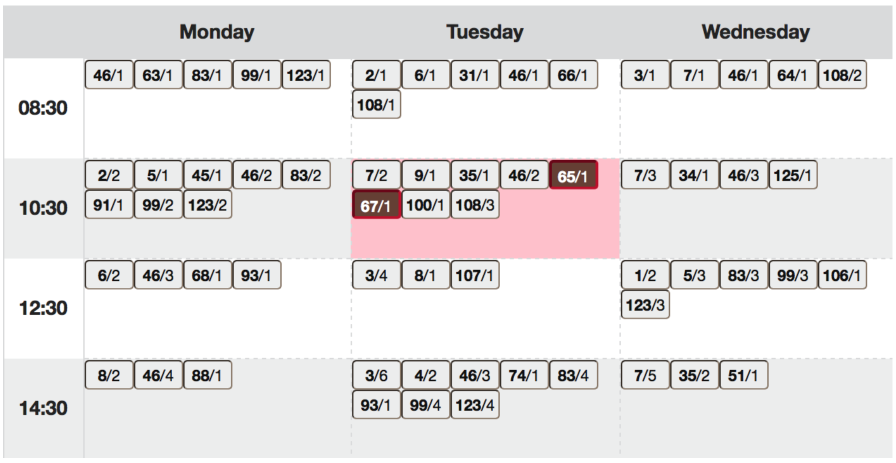
4.2 A Domain Theory of Timetables and Curricula

4.2.1 Background

Timetabling is a family of scheduling or resource allocation problems where the goal is to assign events to a limited number of time slots. Each event might be associated with arbitrary constraints. Corne et al. [36] categorize simple timetabling constraints as unary, binary, capacity, event spread and agent constraints. In the context of educational institutions the timetabling problem is about assigning classes, e.g., lectures and seminars, to time slots. The fundamental binary constraint is that no person should have to attend two events relevant for them at the same time; other constraints might regard room assignment, teacher workload, etc. The timetable *construction* problem is an NP-complete problem [35] where the goal is to find a timetable for a set of events that satisfies a set of given constraints (completely or as many as possible). Timetable *validation* can be understood as a similar problem, where the goal is to decide if a given timetable is feasible with regard to different constraints. From a student's perspective, a timetable in a curriculum is feasible if he or she can attend all units as prescribed by the curriculum in such a way that no constraints are violated and that the degree can be finished within the legal time frame.

Before describing the B validation process in Section 4.3, we now introduce the underlying concepts and relationships in a more general manner.

[1]See `https://www3.hhu.de/stups/prob/index.php/ProB_Java_API` - [Online; accessed 31-March-2017]



	Monday	Tuesday	Wednesday
08:30	46/1 63/1 83/1 99/1 123/1	2/1 6/1 31/1 46/1 66/1 108/1	3/1 7/1 46/1 64/1 108/2
10:30	2/2 5/1 45/1 46/2 83/2 91/1 99/2 123/2	7/2 9/1 35/1 46/2 65/1 67/1 100/1 108/3	7/3 34/1 46/3 125/1
12:30	6/2 46/3 68/1 93/1	3/4 8/1 107/1	1/2 5/3 83/3 99/3 106/1 123/3
14:30	8/2 46/4 88/1	3/6 4/2 46/3 74/1 83/4 93/1 99/4 123/4	7/5 35/2 51/1

Figure 4.1: Graphical representation of a semester week highlighting a detected scheduling conflict. Each numbered box represents an event, the numbers represent the teaching units and groups the events are associated to. The two highlighted units (65 and 67) are in conflict because they are assigned to the same curriculum while their sessions share a time slot on Tuesday at 10:30.

4.2.2 Timetabling Data

The different data types and concepts that compose curricula and timetables are described below. Figure 4.2 shows these entities and how they are associated in a graphical representation, together with an exemplary instance to illustrate the meaning of the different concepts.

- **Courses** is the set of all different university courses available in the data. Each **course** represents a subject, e.g. Computer Science or Biology, it can be either a major or a minor (in some cases it might also be a standalone course).
- **Modules** represent groups of classes that focus on related topics. Each **course** is composed of one or more modules.
- **Levels** are structuring elements used to describe rules about how **modules** are organized in each **course**.
- **Module combinations** are sets of **modules** that represent valid combinations for a **course** according to rules described using **levels**, which is explained in Section 4.2.3.

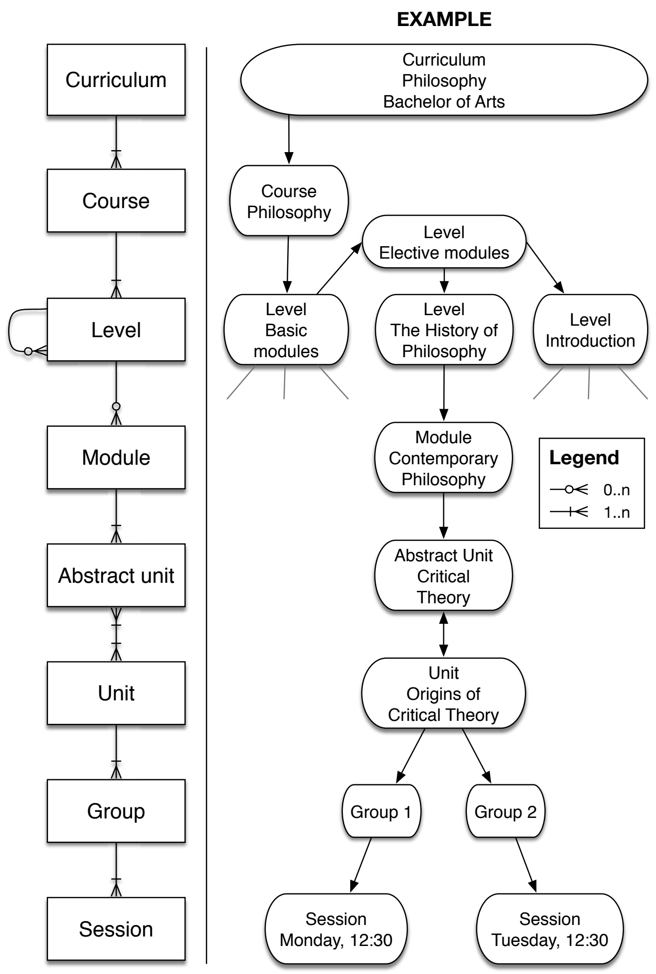


Figure 4.2: Entities and relationships in the third version of our curriculum data model annotated with sample data from Figures 4.3 and 4.9.

- **Abstract units** are associated with a **module** and represent a general topic that should be attended as part of the **module**. This indirection between **modules** and **units**, which we have called **abstract units**, is used to logically group several different **units** that are considered equivalent, within a certain module, from the point of view of the curriculum. Each **abstract unit** can be either *mandatory* or *elective* and is associated with one or more semesters. These semesters represent when, according to the curriculum rules, an **abstract unit** should be attended.
- **Units** represent actual classes as taught at a university. They are linked to **abstract units** and can be interpreted as instances of the **abstract units**. Each **unit** is associated to the semesters in which it is taught. For example a **module** might require an **abstract unit** "Introduction to Formal Methods" which is completed by either visiting the **unit** "Introduction to the B-Method" or "Introduction to Alloy". The actual **units** grouped in an **abstract unit**, might be available in different semesters, cover different, but related topics or be taught by different teachers. To satisfy the requirements of an **abstract unit**, students have to choose and complete one of the **units** available in the **abstract unit**.
- **Groups** are equivalent variations of a **unit**, which are offered at different times and are used for sectioning, where students have to choose one of the groups and thus allowing for smaller groups and more choices regarding the time slots for a class. Each **group** is associated with a **unit** which itself can have one or more **groups**.
- **Sessions** are the different events that take place during the week. They are associated with a day and a time slot when they take place and belong to a **group**, which is composed of one or more **sessions**. Additionally sessions can be scheduled in a weekly or biweekly (in even or odd-numbered weeks) manner.
- The period of the day during which classes are scheduled is divided into blocks of two hours in duration. These blocks are referred to as **time slots**. Each **session** is assigned to exactly one **time slot** on a given day of the week. Assigning **sessions** to **time slots** avoids the problem of partial overlaps or varying class starting times.

The set of all data and relationships either globally or for a specific course is what we will refer to as **curriculum**.

4.2.3 Module Combinations

We have modelled how modules in each course can be combined building upon an existing format provided by the university administration. This format describes the structure of the different curricula as a *tree*, and is reflected in the "Level" data in Section 4.2.2 and Figure 4.2. The trees are composed of **courses**, their **modules** and rules describing the module relationships.

- The leaves of the tree are the **modules**, which are either *elective* or *mandatory*.
- Each inner node is a **level** and is annotated either with the *minimum* and *maximum* number of modules to be chosen for that level or the minimum and maximum number of credits required at each level. These nodes represent logical groupings of modules in the curriculum.
- The root of the tree is a **course**.

The depth of the tree is not fixed beforehand, giving the modeller a lot of flexibility in representing the course structure.

An example of this tree structure is shown in Figure 4.3. The left part of the figure shows the raw format used to represent the trees and the right part shows a graphical representation of the same data.

From Rules to Combinations

The trees described above capture all the modules available in a course and the rules required to decide which modules can be combined. The rules in the levels of a tree must either all be cardinality based or credits based.

Based on these rules a **module combination** can be characterized as follows: Let m be the set of all modules in a given tree t. A module combination mc is a subset of the available modules in t, such that $mc \subseteq m$ and mc satisfies the following predicates:

$$\forall_l \in t \bullet level\_mandatory\_modules(l) \subseteq mc \tag{4.1}$$

Equation (4.1) expresses that for each level l in a tree t the mandatory modules reachable from l must be in the set mc of chosen modules. *level_mandatory_modules* in Equation (4.1) is a function that maps from each level to the mandatory modules in its subtree.

For cardinality based rule-trees a module choice mc must additionally satisfy constraint, shown in Equation (4.2), that for each level l in the tree t the number of modules selected in the subtree reachable from l satisfies the cardinality constraint for l, where *level_available_modules* is the function that associates each module with the set of modules reachable from it its subtree and *level_min* and *level_max* are the functions that associate a given level with the corresponding minimum and maximum numbers of required modules.

$$\forall_l \in t \bullet level\_min(l) \leq \mid level\_available\_modules(l) \ \cap \ mc \mid \leq level\_max(l) \tag{4.2}$$

In credit-point based rule-trees a module choice mc must satisfy the similar predicate, shown in Equation (4.3), ensuring that the sum of credit points awarded for the modules selected in the subtree of l is in the required range according to l. Where $lm(l, m) = level\_available\_modules(l) \ \cap \ m$ and *level_min_cp* and *level_max_cp* are functions that associate each level with the minimum and maximum number of credit points to be collected in the modules selected for a given level.

$$\forall_l \in t \bullet level\_min\_cp(l) \leq \sum_{m \ \in \ lm(l,mc)} credit\_points(m) \leq level\_max\_cp(l) \tag{4.3}$$

For the root node of a tree all combinations of results valid for its children represent valid **module combinations** for that **course**.

For example, the tree in Figure 4.3 describes how to choose modules for a fictional "Philosophy" course.

- The topmost "Elective Modules" level stipulates that we need to choose three to six modules. Some of the inner levels, however, impose additional constraints on this choice.
- The "Introduction" level stipulates that we must choose one or two modules.

```
...
<course name="Philosophy">
  <level name="Elective modules" min="3" max="6">
    <level name="Introduction" min="1" max="2">
      <module name="Logic I" mandatory="t" pordnr="1"/>
      <module name="Basic Concepts" pordnr="2"/>
    </level>
    <level name="The History of Philosophy" min="2" max="4">
      <module name="Antiquity" pordnr="3"/>
      <module name="Medieval Philosophy" pordnr="4"/>
      <module name="Modern Philosophy" pordnr="5"/>
      <module name="Contemporary Philosophy" pordnr="6"/>
    </level>
  </level>
  ...
</course>
...
```

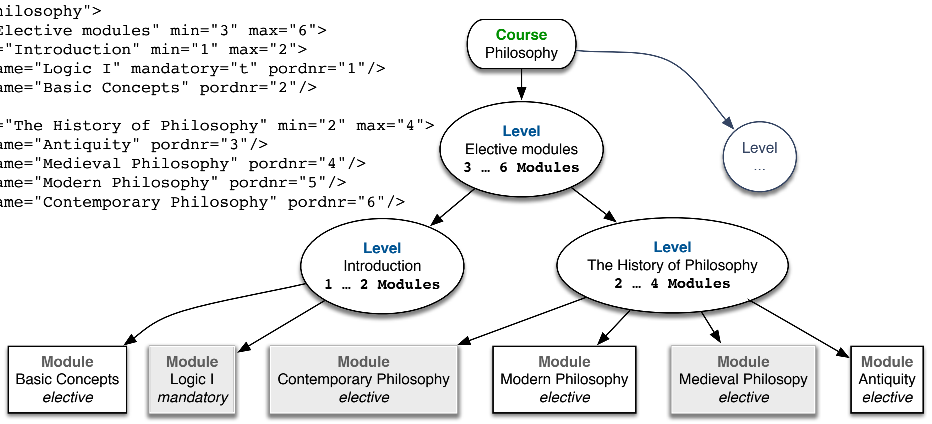


Figure 4.3: Modules and module selection rules for an exemplary course "Philosophy". Rules and modules are shown in the raw XML data format and as an equivalent graphical representation, where one of the valid module combination is highlighted.

- The "Logic I" module, since it is marked as mandatory, must be chosen as part of the "Introduction" level.
- The "History of Philosophy" level stipulates that we have to take between two and four modules within that area.

One out of the twenty-two valid choices of modules (shown in Table 4.1) for this tree is to select the modules "Logic I", "Contemporary Philosophy" and "Medieval Philosophy" (highlighted in Figure 4.3). An instance of an invalid choice are the modules "Basic Concepts", "Logic I" and "Contemporary Philosophy". Although this choice satisfies the requirements for the "Introduction" and "Elective modules" levels, it is invalid because we have only chosen one module for the level "The History of Philosophy" despite the fact that it requires at least two.

Table 4.1: All possible module combinations for the module tree in Figure 4.3.

n	Modules					
1	Logic I	Antiquity	Contemporary Ph.			
2	Logic I	Antiquity	Medieval Ph.			
3	Logic I	Antiquity	Modern Ph.			
4	Logic I	Contemporary Ph.	Medieval Ph.			
5	Logic I	Contemporary Ph.	Modern Ph.			
6	Logic I	Medieval Ph.	Modern Ph.			
7	Logic I	Antiquity	Contemporary Ph.	Medieval Ph.		
8	Logic I	Antiquity	Contemporary Ph.	Modern Ph.		
9	Logic I	Antiquity	Medieval Ph.	Modern Ph.		
10	Logic I	Contemporary Ph.	Medieval Ph.	Modern Ph.		
11	Logic I	Antiquity	Basic Concepts	Contemporary Ph.		
12	Logic I	Antiquity	Basic Concepts	Medieval Ph.		
13	Logic I	Antiquity	Basic Concepts	Modern Ph.		
14	Logic I	Basic Concepts	Contemporary Ph.	Medieval Ph.		
15	Logic I	Basic Concepts	Contemporary Ph.	Modern Ph.		
16	Logic I	Basic Concepts	Medieval Ph.	Modern Ph.		
17	Logic I	Antiquity	Contemporary Ph.	Medieval Ph.	Modern Ph.	
18	Logic I	Antiquity	Basic Concepts	Contemporary Ph.	Medieval Ph.	
19	Logic I	Antiquity	Basic Concepts	Contemporary Ph.	Modern Ph.	
20	Logic I	Antiquity	Basic Concepts	Medieval Ph.	Modern Ph.	
21	Logic I	Basic Concepts	Contemporary Ph.	Medieval Ph.	Modern Ph.	
22	Logic I	Antiquity	Basic Concepts	Contemporary Ph.	Medieval Ph.	Modern Ph.

4.2.4 Curricula/Timetables

Based on the entities and relationships described above we will now define the concepts of syllabi, timetables and feasibility of timetables. For this we use the feature diagram notation known from basic and cardinality based feature models [75, 37], as it is perfectly suited to capture the constraints and relationships present in our curricula.

Feature Modelling

Feature models are a way of describing a system or a family of systems by means of their features and variations. These features and their constraints are organized in a hierarchical structure, providing a language to describe how features are related and how they can vary in different instances of a system. Feature models were originally introduced by Kang et al. [75] and are frequently used in the context of software product lines (SPL).

In the hierarchical structure of feature models, sub-features can be designated as **mandatory** or **optional**. Additionally there can be **or** and **xor** relations used to describe configurations for groups of sub-features. *Feature diagrams* are a common graphical notation for feature models to capture the features and their relationships. There are two types of feature diagrams, basic and cardinality based [37]. Basic feature diagrams capture features and their variability, while in cardinality based feature diagrams the association between features can be extended with a cardinality constraint on the number of instances a feature should have.

In a feature diagram, features are denoted by the nodes in the diagrams and the different associations, also shown in the legend of Figure 4.4, are as follows:

- **Mandatory** sub-features are denoted by a line ending in black circle and these have to be selected if the parent node is selected in order to represent a valid configuration.
- **Optional** features are represented using a white circle.
- **Alternative** sub-features, corresponding to the logical `xor` operator are denoted by a white arc connecting the edges that go from the parent feature to the sub-features.

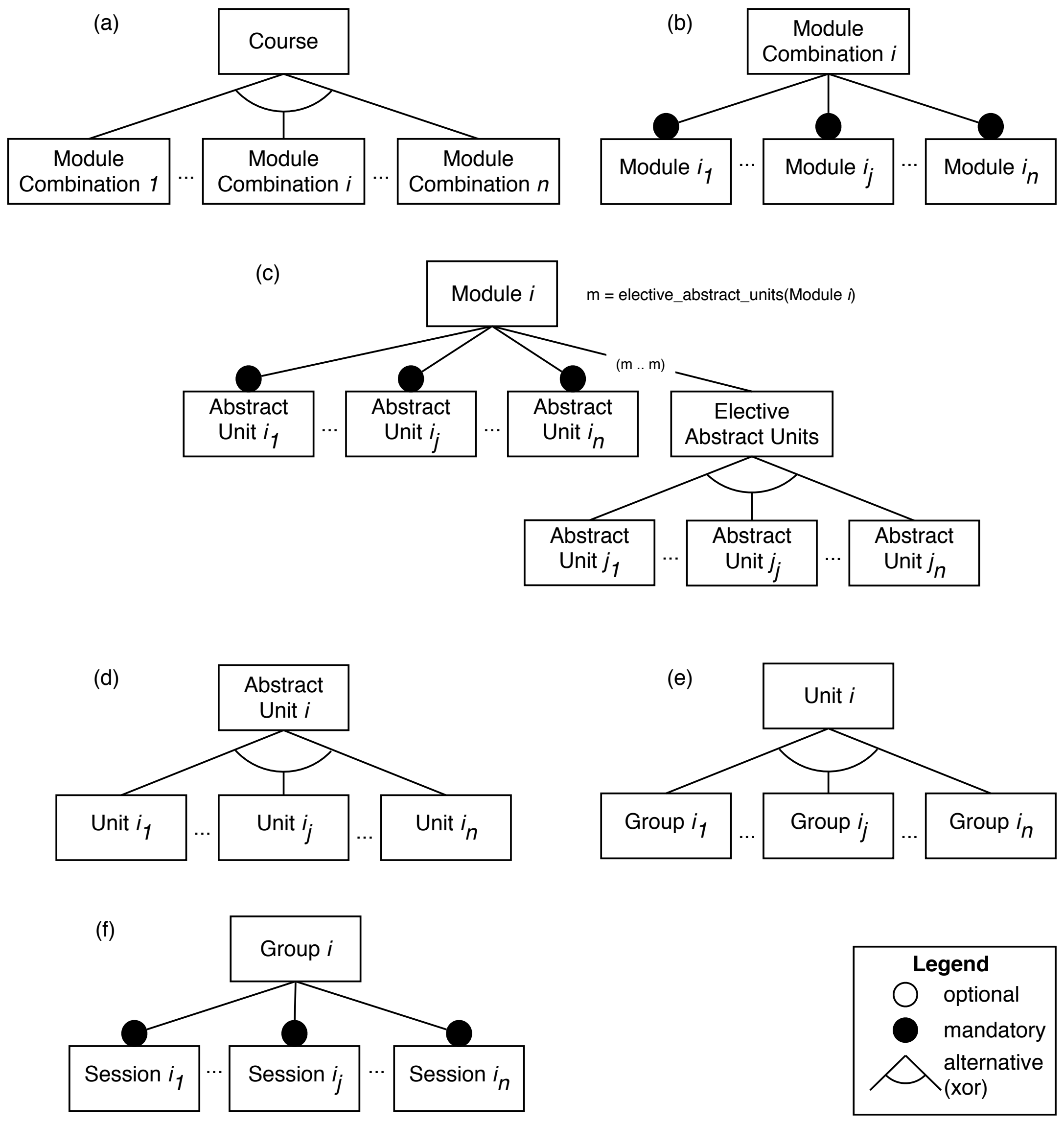


Figure 4.4: Feature diagram describing the different selection rules in a curriculum.

- The **or** association is analogously denoted by a filled black arc connecting the edges from the parent feature to the sub-features that are or-associated.
- **Cardinality** constraints are an annotation on the edges of the form `m..n` denoting the minimum and maximum number of allowed instances of a feature.

A **product** in this context is any valid selection of features that satisfies the constraints imposed by the feature model.

Feature Model of Feasible Timetables

The rules, visualized as feature diagrams in Figure 4.4, for creating a valid choice of modules, abstract units, etc. are as follows:

- For each **course** one of the previously described **module combinations** is to be selected, as shown in Section (a). Each of these is a set of **modules**.
- In a given **module combination** all **modules** are mandatory, Section (b) captures this rule.
- **Modules**, as described above, are composed of **abstract units**. In any selected **module**, as can be seen in Section (c), all mandatory **abstract units** have to be selected. The number of elective **abstract units** that have to be selected in each module is is represented by the function *elective_abstract_units* : **Modules** $\rightarrow \mathbb{N}$.
- For each selected **abstract unit**, as shown in Section (d) of the figure, exactly one **unit** that implements it has to be chosen.
- For each selected **unit** one **group** must be selected, see Section (e).
- In each **group** all **sessions** are mandatory, as shown in Section (f).

The combination of all diagrams in Figure 4.4 yields a feature diagram representing all possible ways of completing a course for a given data set.

A **syllabus**, which is the solution we are looking for, is a choice of modules, abstract units, units, groups and sessions according to the rules of the curriculum, i.e. a product in the feature model vocabulary.

Feasible Timetables

Each **syllabus**, i.e. each product in the feature model described above, represents a selection of classes that together satisfy the requirements for a course's curriculum such that we can build a timetable for it.

The relations between the elements of the **syllabus** are described by the following predicates:

- $group\_choice(u, g)$ represents that **group** g was selected for **unit** u in the syllabus.
- $unit\_choice(au, u)$ represents that **unit** u was selected for **abstract unit** au in the syllabus.

Additionally, we have modelled the following relations to describe the associations present in the curriculum data:

- $group\_session(g, s)$ relates **group** g and **session** s, i.e. s is a **session** in **group** g.
- $session(s, d, t, r)$ represents a **session** s scheduled on a day d at time t, with rhythm r. Where r is one of *weekly*, *biweekly_even* or *biweekly_odd*.
- $abstract\_unit\_semester(au, s)$ relates **abstract unit** au with semester s, i.e. s is a semester (of possibly several alternatives) where au should be attended according to the curriculum rules.
- $unit\_semester(u, s)$ is the analogous predicate that relates a **unit** u with a **semester** s (of possibly several) where it is taught.

A **timetable** is a **syllabus** extended by the relation *semester_choice*. This relation assigns to each selected abstract unit in a **syllabus** one of the semesters recommended for it in the curriculum. The **timetable** has the additional constraint that each selected unit has to be taught in the semester selected for its corresponding abstract unit.

- $semester\_choice(au, s)$ represents that in the syllabus **semester** s was selected for **abstract unit** au.

To illustrate, let *selected_abstract_units* be the set of abstract units in the **syllabus** then, for any **timetable** the following must be true:

$$
\begin{aligned}
\forall_{au} \in \mathit{selected\_abstract\_units} \bullet \exists_{s,u} \bullet\ & s \in 1 \ldots 6 \\
& \wedge \mathit{abstract\_unit\_semester}(au, s) \\
& \wedge \mathit{semester\_choice}(au, s) \\
& \wedge \mathit{unit\_choice}(au, u) \\
& \wedge \mathit{unit\_semester}(u, s)
\end{aligned}
$$

Not every possible timetable is actually feasible from a student's perspective. A **feasible timetable** is a timetable as defined above that satisfies the additional constraint, that it is free of binary conflicts among those events that are assigned to the same semester by the timetable as described above.

We can define the **feasibility** in terms of the absence of binary conflicts in the **timetable**.

Conflicts are based on the **sessions** in a **group** chosen for a **unit**. **Sessions** have a day and a time field that together represent the time slot for the session and, a session can take place weekly or biweekly (either in even or odd numbered weeks) which is represented by a rhythm field with possible values *weekly*, *biweekly_even* and *biweekly_odd*. From the conflict property for a pair of **sessions** we can infer the conflict property for **groups**, **units** and **abstract units** as follows:

Two **sessions** are in conflict if the **sessions** are scheduled on the same **time slot** and on an interfering rhythm (e.g., at least one weekly or both on the same biweekly rhythm).

$$
\begin{aligned}
\mathit{session\_conflict}(s_1, s_2) \equiv \exists_{d,t,r_1,r_2} \bullet\ & \mathit{session}(s_1, d, t, r_1) \wedge \mathit{session}(s_2, d, t, r_2) \\
& \wedge (r_1 = r_2 \vee r_1 = \mathit{weekly} \vee r_2 = \mathit{weekly})
\end{aligned}
$$

Related **sessions** form a **group**; a **group** is in conflict with another **group** if there is a **session** in each **group**, such that these are in conflict.

$$\begin{aligned} \mathit{group\_conflict}(g_1, g_2) \equiv \exists_{s_1,s_2} \bullet\ & \mathit{group\_session}(g_1, s_1) \\ & \wedge \mathit{group\_session}(g_2, s_2) \\ & \wedge \mathit{session\_conflict}(s_1, s_2) \end{aligned}$$

A pair of **units** is in conflict if the **groups** chosen for each in the **timetable** are in conflict.

$$\begin{aligned} \mathit{unit\_conflict}(u_1, u_2) \equiv \exists_{g_1,g_2} \bullet\ & \mathit{group\_choice}(u_1, g_1) \\ & \wedge \mathit{group\_choice}(u_2, g_2) \\ & \wedge \mathit{group\_conflict}(g_1, g_2) \end{aligned}$$

A pair of **abstract units** is in conflict if they have been assigned to the same semester in the **timetable** and the selected **units** are in conflict.

$$\begin{aligned} \mathit{abstract\_unit\_conflict}(au_1, au_2) \equiv \exists_{s,u_1,u_2} \bullet\ & \mathit{semester\_choice}(au_1, s) \\ & \wedge \mathit{semester\_choice}(au_2, s) \\ & \wedge \mathit{unit\_choice}(au_1, u_1) \\ & \wedge \mathit{unit\_choice}(au_2, u_2) \\ & \wedge \mathit{unit\_conflict}(u_1, u_2) \end{aligned}$$

If all pairs of **abstract units** for the **module combination** in the **timetable** are free of conflicts, then the **timetable** is **feasible** from a student's perspective. This means that there is at least one way for students to choose **modules** and schedule all **units** for those modules without binary conflicts. This form of validation leaves out of consideration, how the students are distributed on the different classes and that there might be certain classes available, but practically impossible to attend, as they are scheduled at the same time as a mandatory class. Such a scheduling conflict is acceptable from a feasibility point of view, as long as there are viable alternatives in the timetable data.

4.3 Modelling Curricula and Timetables in B

Having introduced the core concepts and abstractions of the domain this section focuses on the central and non-trivial features of our application, in particular those aspects we have modelled formally in B. Our goal is to explore the usability of B for complex constraint modelling and solving applications while creating a case study to drive further enhancements to the constraint-solving features in ProB, that should benefit all users of the ProB tool.

There are several aspects to discuss concerning modelling and validating timetables, e.g., importing data into a model, deriving information from the data, etc. The structure of this section, described below, discusses our modelling based on the different steps involved in the validation process.

1. The first step in the process is to import data from user provided formats to a representation usable in a B model, which is described in Section 4.3.1.
2. From the imported data we compute several derived values used in the validation process, as explained in Section 4.3.1.
3. Based on the imported data and derived values we can validate curricula when requested by a user following the schema outlined in Section 4.3.2.
4. If the validation logic detects a feasibility conflict, in Section 4.3.4 we describe how we compute a minimal conflict set of the timetable data to help users identify conflicts.
5. To solve a conflict or to move a sessions without introducing new conflicts we have modelled how to find alternative time slots, which is discussed in Section 4.3.5.
6. In Section 4.3.6 we discuss how the features discussed before are organized in a B model.
7. Lastly, in Section 4.3.7 we provide some details on how the models and data have evolved in the course of the project.

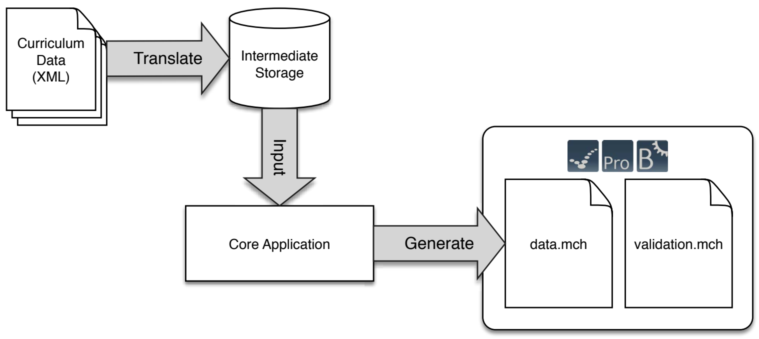


Figure 4.5: Schematic representation of the data import and transformation process: the user provided raw data is translated into an intermediate representation that serves as input for the main application. The main application generates a B representation of the data which together with the validation machine is used to check the feasibility of the data.

4.3.1 Data Import & Representation

The data to be validated is collected and provided by the participating faculties. The data representation and the steps taken to integrate the data into the validation model have evolved over time (see Section 4.3.7 for details). During the project we have stepwise refined the representation of the curriculum data and have extended the number of characteristics covered in our models. In general the structure how our curricula are organized, is shown in Figure 4.2.

In the current version of our tool we use the flexible, hierarchical representation of courses, with arbitrary nesting of levels, as described in Section 4.2.3. This allows us to capture more detailed and complex structures. Furthermore, we are using a single shared data representation for all participating faculties. The data is provided as XML documents by the faculties following the shared schema and processed for use in combination with our validation model.

From the provided files we derive an intermediate representation, currently stored in a database. In a second step, this is used to generate a B machine containing the curriculum data, as outlined in Figure 4.5. How the intermediate storage is integrated into the application is discussed in Section 4.4.

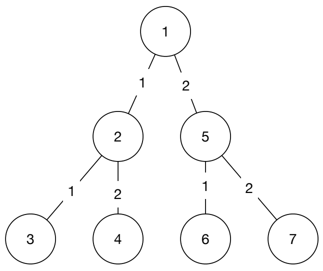


Figure 4.6: Simple tree structure with labelled edges representing the order of the child nodes.

Curriculum Rules

Curriculum rules, as described in Section 4.2.3, are represented as trees of an arbitrary depth. Initially, we mapped these trees of levels and modules to the B pattern of representing trees as a function of sequences and values [3]. Each sequence represents a path within the tree and maps to the value of the corresponding node. Based on the Atelier-B [31] handbook, we have added support for the B tree operators to ProB (e.g., `father` and `subtree`). One drawback of this representation of tree is the overhead associated with operations that return parts of the original tree, since all path sequences have to be recomputed with regard to the new root of the returned tree.

Originally, we represented a selection of modules as a bitvector. By traversing the tree of curriculum rules we would validate the configuration represented by the bitvector against the level- and module-nodes. Due to the large number of possible module combinations, the overhead associated with the tree operations, and to avoid recomputing the combinations each time the B representation of the data is loaded, we later decided to replace this approach. Instead, now when we import the data we precompute all valid module combinations and store them in our intermediate representation.

The Atelier-B [31] handbook defines several operators to work with trees. Trees are created using the `bin` and `const` operations for leaves and inner nodes respectively.

As an example, the simple tree shown in Figure 4.6 would be defined as B using the `bin` and `const` tree operators as shown in Figure 4.7.

```
>>> const(1, [const(2, [bin(3), bin(4)]), const(5, [bin(6),
   bin(7)])])

        {([] ↦ 1), ([1] ↦ 2), ([1,1] ↦ 3), ([1,2] ↦ 4),
                  ([2] ↦ 5), ([2,1] ↦ 6), ([2,2] ↦ 7)}
```

Figure 4.7: Construction of a tree data structure in B.

This constructs a tree where `[]` denotes the root and `[1,2]` represents the path from the root node to the first child and from there to the second child with the value 4.

Module Combination Validation The B implementation of the module combination computation is not efficient enough to be used in the application. Nevertheless, a B specification of the curriculum rules is useful to validate the external computation by comparing the results produced by both implementations and to illustrate how the combinations are computed.

We describe the trees of levels, that represent the module selection rules, as sets of sequences using the operators mentioned above. The data for each level (e.g. level title, minimum and maximum number of modules to be chosen for that level, etc.) is stored as a record on the corresponding node in the tree. The variables `level_1` and `level_2` in Figure 4.8 represent an inner node and a leaf of such a tree.

Each course is associated with the root level of their corresponding tree of levels (`course_levels` in Figure 4.8). On the other hand, the leaf nodes are linked to the modules they contain (`level_modules` in Figure 4.8).

The procedure to compute the module combinations in B is as follows (the B machine used to validate module combinations can be found in Appendix C). Each module combination is represented using a bitvector, which itself is represented using a total function from the set of collected modules to `BOOLEAN`. First we collect all trees and all available modules for a given course. Each bitvector is stepwise constrained with regard to the currently evaluated tree and modules. All modules that are mandatory for the current course are mapped to `TRUE` in the bitvector, since they must be part of any valid combination of modules. Conversely, all modules that are not associated to the course, are required to be mapped to `FALSE`, since the cannot be contained in any valid combination of modules.

```
level_1 = const(rec(idx:level1, name:"Elective modules",
                from:3, to:6), [level_2,level_3])
&
level_2 = bin(rec(idx:level2, name:"Introduction", from:1, to:2))
...
course_levels = { ("Philosophy", level_1), ...}
...
level_modules = { ...(level2, mod1), (level2, mod2),
                                  (level3, mod3), ...}
```

Figure 4.8: B representation of level trees.

Further constraints are derived by evaluating the different level-nodes in the trees. Valid module combinations are constrained to those that satisfy the rules for each level. This means that for each level we compute, as shown in Fig. 4.9, the set of reachable modules in its subtree. For each level we constrain the bitvector in two ways. First, only modules reachable from the current level are allowed to be selected. Second the number of selected modules in the current subtree must be within the limits specified in the current level. This is expressed by computing the set of modules selected in the current subtree, see Fig. 4.9, and constraining the cardinality of the set to be within the `from` and `to` limits of the current level.

```
llmm = level_available_modules(level_info'idx)
...
/* the number of modules chosen at each level
   must be in the limits of from .. to */
card({y| y : mm & y ↦ TRUE : choice(tt) &
        y : llmm}) : level_info'from .. level_info'to
```

Figure 4.9: Validation step to check that the number of selected modules for a level-node is within the permitted bounds.

Credit-point based module combinations work generally the same way, only that instead of considering the number of selected modules we consider the sum of each selected module's credit points.

Curriculum Data

The second part of the data captures the *content* of the courses, corresponding to the lower half of Figure 4.2. The content of the courses is composed of:

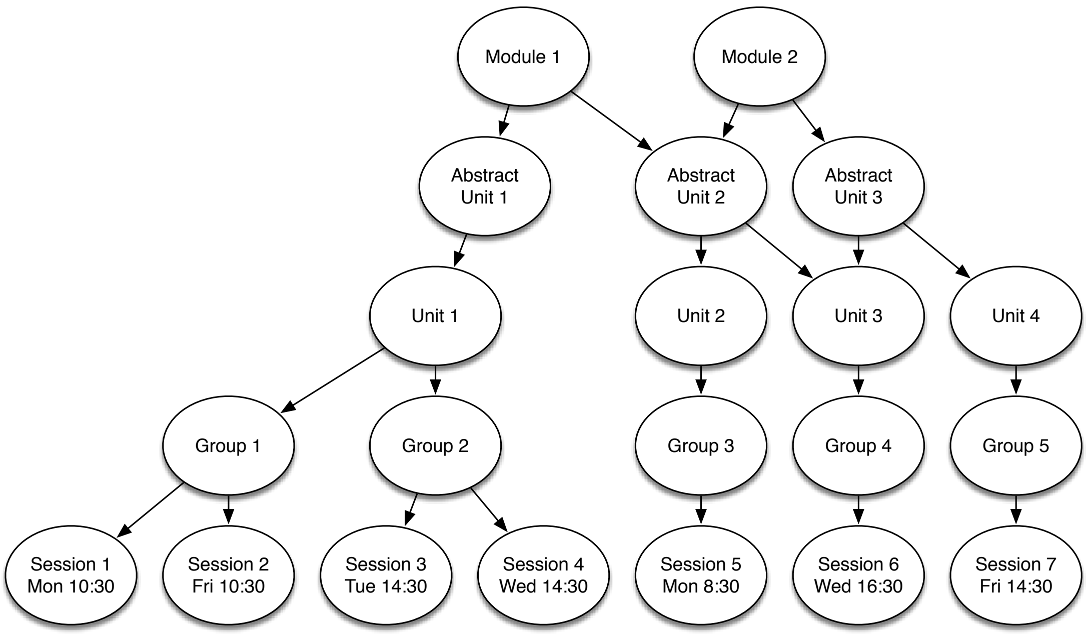
- the different **modules** available in the curriculum.
- Each **module** is composed of **abstract units**. These can be mandatory or elective and carry the information in which semester they should be attended.
- Each **abstract unit** is linked to one or more **units** available in the **curriculum**.
- Each **unit** has the information in which semester it is taught and has one or several **groups** for students to choose.
- Each **group** is composed of one or many **sessions** the students should attend, **sessions** have the information on which day and time they take place and in which rhythm they take place (weekly or biweekly).

Although each **unit** can be linked to more than one **abstract unit**, it only counts towards the completion of one. So, if a **unit** is linked to two **abstract units**, students still have to choose different units for the abstract units.

Figure 4.9 shows the different entities and how they are connected. In an example "Module 1" contains "Abstract Unit 1" and "Abstract Unit 2". In order to complete the latter students can choose if they want to attend "Unit 2" or "Unit 3" in order to satisfy the requirement. Choosing "Unit 3" would satisfy the requirements for "Abstract Unit 2" or "Abstract Unit 3" but not both at the same time.

```
<modules>
  <module name="module 1">
    <abstract-unit id="au1" title="abstract unit 1" />
    <abstract-unit id="au2" title="abstract unit 2" />
  </module>
  <module name="module 2">
    <abstract-unit id="au2" title="abstract unit 2" />
    <abstract-unit id="au3" title="abstract unit 3" />
  </module> ... </modules>
<units>
  <unit title="unit 1">
    <group title="group 1">
      <session title="session 1" day="mon" time="10:30" />
      <session title="session 2" day="fri" time="10:30" />
    </group>
    <group title="group 2">
      <session title="session 3" day="tue" time="14:30" />
      <session title="session 4" day="wed" time="14:30" />
    </group>
    <abstract-unit id="au1" />
  </unit>
  <unit title="unit 2">
    <group title="group 3">
      <session title="session 5" day="mon" time="8:30" />
    </group>
    <abstract-unit id="au2" />
  </unit>
  <unit title="unit 3">
    <group title="group 4">
      <session title="session 6" day="wed" time="16:30" />
    </group>
    <abstract-unit id="au2" />
    <abstract-unit id="au3" />
  </unit>
  <unit title="unit 4">
    <group title="group 5">
      <session title="session 7" day="fri" time="14:30" />
    </group>
    <abstract-unit id="au1" />
  </unit> ... </units>
```

(a) XML representation of modules, abstract units, units, sessions and groups and their relationships



(b) Graphical representation of modules, abstract units, units, sessions and groups and their relationships

Figure 4.9: XML and graphical representation of entities and relationships in the curriculum data.

```
sessions : POW(SESSIONS * struct(duration: INTEGER, rhythm: RHYTHMS,
                                 dow: DAYS, time: SLOTS)) &
sessions = {(session1 ↦ rec(duration: 2, rhythm: weekly,
                                 dow: monday, time: 2)), ...} &

groups : POW(GROUPS * struct(sessions: POW(SESSIONS))) &
groups = {(group1 ↦ rec(sessions: {session1, ...})), ...} &

units : POW(UNITS * struct(idx: INTEGER, title: STRING,
                             semesters: POW(SEMESTERS), groups:
                                POW(GROUPS))) &
units = {(unit1 ↦ rec(idx: 1, title: "Lecture",
                             semesters: {sem1, sem2}, groups:
                                {group1})), ...}
```

Figure 4.10: Representation of sessions, groups and units as B records.

Data Representation

From the preprocessed data, stored in a database, we generate a B model to be used by our validation model (Figure 4.5). In this model we represent each of the described entities (units, courses, etc.) as B records, and generally each database field as a field in the record. It is often simple to map an entity in the source data to a record in B, grouping all attributes in one location. Using records it is easy to access each attribute of an entity using the quote operator for field access, in particular when compared to tuples (which would have been another B construct to group all attributes of an entity).

Sets of values are mapped to functions in the `PROPERTIES` section of the generated machine. Each function maps from a unique identifier, i.e. derived from the database identifier of the entry, to the corresponding B record. To illustrate, an entry `<unit title="Lecture" ... />` in the original data, assuming it was assigned the id 1, would be mapped in the `units` function to `(unit1 |-> rec(title: "Lecture", ...))`.

Relations are represented using the identifiers of the records. E.g. a unit with one group and one session might be represented as shown in Figure 4.10. Many to many relations are represented as sets of tuples. Each tuple contains the identifiers of the referenced items and might contain additional information associated to the relation. E.g. the

```
 /* set of all mandatory abstract units */
mandatory_abstract_units = {mm | mm : ABSTRACT_UNITS &
                              abstract_units(mm)'type = "mandatory"}
```

Figure 4.11: Computing the set of IDs for all mandatory abstract units according to their type field.

combinations of module, abstract unit and recommended semesters for the abstract unit in that module would be represented by several triples of module, abstract unit and semester.

Besides the source data, the precomputed combinations of modules for each course are translated into a set containing sets of module identifiers. Each of these sets represents one valid choice of modules according to the curriculum rules as described above.

Derived Data

When our validation model is loaded by ProB it includes the generated data model. During the initialization of the model we compute several values that are derived from the values in the data machine.

Deriving properties from the raw data is useful for several reasons, first it allows us to compute and store values that are needed frequently in the validation process, avoiding to recompute them every time. Second we can compute information about relationships present in the data, that are not explicitly encoded in the data representation. Third we can compute certain static properties of the data that can be used to decide the feasibility of courses statically, e.g., if there are no abstract units for a mandatory module it means that the corresponding course is never feasible. These derived values are computed as part of the setup of the validation machine, in the `PROPERTIES` section and stored as `CONSTANTS` for later use.

As an example `mandatory_abstract_units` in Figure 4.11 is a set that contains the identifiers for all mandatory abstract units, collected from the `abstract_units` function which maps each identifier to the corresponding record for each abstract_unit.

In this part of the modelling process, we actually use B as a functional programming language. On the positive side, B provides quite a few useful constructs to apply functions. Take for example the squaring function defined by `sqr = %(x).(x : INTEGER | x*x)`. Given that definition we can for example:

- map the function of a sequence s of values using relational composition: `([1,2,3] ; sqr) = [1,4,9]`,
- compute the image of the function for a set of values: `sqr[1..3] = 1,4,9`,
- compose functions using again relational composition: `(sqr ; sqr) (2) = 16`.

On the downside, one cannot define polymorphic functions in B. Furthermore, B has no built-in support for let-constructs and if-then-else at the expression level. Defining recursive functions is thus cumbersome (see Section 4.3.4 and Figure 4.14). To alleviate this, ProB now supports the use the let-constructs and if-then-else syntax in expressions and predicates, which will be discussed in chapter Chapter 5. The lower half of Figure 4.14 shows a function written using the if-then-else substitution syntax now available in predicates and expressions.

4.3.2 Validation

The goal of our tool is to detect if a given curriculum is feasible. The core criterion to decide the feasibility of a curriculum is the presence or absence of binary session conflicts from a student's perspective. A binary conflict occurs if two events are scheduled at the same time. In our case there is a binary conflict if two sessions from two selected units are scheduled for the same day and time. Additionally, there are a few restrictions on this rule, as mentioned in Section 4.2.2, sessions can be scheduled on a weekly or biweekly rhythm (a distinction is made between even and odd-numbered weeks) Sessions scheduled in a non-interfering rhythm can never be in conflict, even though they share the same time slot.

Our approach to detect if an input is free of binary conflicts is to first, for a given curriculum, find a valid choice of units, as described in Section 4.2.4. For a choice of units we then try to decide if it is possible to attend all of them without binary conflicts by trying to find a group and a semester when to attend each. We have split this process into two parts. The first is only concerned with finding sets of abstract units and their units that might represent a feasible combination. The second does not need to take into account how the course being validated is organized and is only concerned with deciding if a given set of units (computed in the first step) is free of conflicts.

The first part is concerned with finding a choice of units that satisfies the requirements of the curriculum. The specific rules on how to choose units might differ for each faculty, e.g., one might require a choice based on the number of units attended another might require a specific amount of credit points in each module, etc. The general process is:

- Starting from a course (or courses for a combination of major and minor) we select one of the precomputed valid choices of modules, each composed of all *mandatory* and a subset of *elective modules.*

- In each module we collect all *mandatory abstract units* and a subset of the *elective abstract units* according to the curriculum's and module's rules. E.g., a module might require students to take one mandatory and two out of five elective abstract units.

Based on the set of abstract units selected we define a function `unitChoice` using the abstract units as its domain and mapping to the set of units. The range of the function is constrained based on the units associated to each abstract unit and ProB will instantiate the function at runtime to find a valid instance. The set of all selected units, the range of `unitChoice` represents one possible combination of units sufficient to obtain a degree. Next, we want to detect if this specific choice is not only sufficient but also feasible.

The second part of the validation logic, shown in Figure 4.12, is a predicate concerned with the decision if a given choice of units is free of conflicts. That is, to find a semester in which to attend each unit and to choose a group in each such that no two sessions are at the same time during the same semester. We have modelled this by defining two total functions, `semesterChoice` and `groupChoice`, both chosen by ProB – at runtime, based on the provided constraints – from the set of functions that map from the collected abstract units to the available semester and from the units to the available groups respectively, the definition of `groupChoice` is shown in Figure 4.13.

```
!(au1, u1, au2, u2).(((au1, u1) : unitChoice &
    (au2, u2) : unitChoice &
    u1 /= u2 & au1 /= au2 &
    semesterChoice(au1) = semesterChoice(au2) &
    semesterChoice(au1) : units(u1)'semesters &
    semesterChoice(au2) : units(u2)'semesters)
      ⇒ #(group1, group2).(
            group1 = unit_group(u1, groupChoice(u1)) &
            group2 = unit_group(u2, groupChoice(u2)) &
            !(s1, s2).(s1 : group1'sessions & s2 : group2'sessions
              ⇒ ((s1'rhythm = s2'rhythm /* both in the same rhythm */
                or s1'rhythm = weekly /* first weekly */
                or s2'rhythm = weekly) /* second weekly */
                & s1'dow = s2'dow) /* same day of week */
                    ⇒ s1'time /= s2'time)))
```

Figure 4.12: Simplified conflict detection logic for a choice of units.

We constrain the ranges of these functions based on the provided data, e.g. in which semester a specific unit should be attended is used to constrain the possible choices for a semester. And we enforce the constraint, the selected unit must be available in the selected semester.

The conflict-property can be expressed in B as a (nested) universally quantified predicate over the set of pairs of abstract and concrete units. Conflicts are based on the sessions of the chosen group for a unit. Sessions have a day and a time field that together represent the time slot for the session. From the conflict property for a pair of sessions we can infer the conflict property for groups as previously described in Section 4.2.4.

```
groupChoice : UNITS → min_group .. max_group &
     !(u).(u : UNITS
         ⇒ groupChoice(u) : unit_min_group(u) .. unit_max_group(u))
```

Figure 4.13: Defining the choice of group constraint using a total function (`groupChoice`).

One of the challenging aspects for ProB to solve this problem is finding valid instances of the `unitChoice`, `semesterChoice` and `groupChoice`. The formula in Figure 4.12 is only true if these functions can be found such that they satisfy the given constraints and lead to no binary conflicts among sessions. If the validation fails ProB will try a different instantiation of these functions until it either finds one that satisfies the provided constraints or there are no further possible choices.

4.3.3 Assisting the Solving Process

There are certain configurations, where the ProB's constraint solver is not powerful enough to detect that there is no solution to a query. In many cases this is due to a combination of missing data that leads to a validation failure and many possible choices in the present data, e.g. many module combinations that can be tried. These situations can lead to ProB trying all possible configurations and failing.

Detecting Statically Infeasible Courses We have extended our validation by computing, as a derived property of our machine, a set of courses that due to the structure of the provided data are known to be infeasible. This means, that it is impossible to choose a combination of modules, abstract units, etc. which is feasible. This is the case because, although the available events might be free of binary conflicts, the provided data does not satisfy the constraints about the structure of the data.

Some of these static validations we currently perform are:

- Mandatory abstract units that contain only one unit are impossible to complete if the unit is the same in both. The two abstract units would require choosing the same unit in both are thus impossible to select side by side, since this would require at least a second unit.
- Mandatory abstract units with no assigned units, make a course impossible to complete. Mandatory abstract units have to be attended, but they cannot be completed if there are no associated units in the data.
- Abstract units contain recommended semesters in which they should be completed. If none of the associated units is taught in one of the recommended semesters it is impossible to find a solution. If an abstract unit a should be attended in the first two semesters, but the associated unit is only taught in the third semester there is no way to find a timetable for the course that satisfies a's semester requirement.

4.3.4 Computing Conflict Sources

If the validation process does not find any conflicts it will generate an assignment to the `unitChoice`, `semesterChoice` and `groupChoice` functions. Together these represent one of possibly many feasible syllabi for the curriculum being validated. If no assignment can be found, the curriculum contains feasibility conflicts. Thus it is necessary to find and resolve any scheduling conflicts among the units in the curriculum before generating a viable timetable.

To identify which units cause a conflict we compute a *minimal unsatisfiable core* (UC) [96, 124] of the units used in the validation. Note that we do not want to compute the unsatisfiable core of the constraints, but rather the unsatisfiable core of the data, i.e., a minimal set of units that cause a conflict.

To compute the unsatisfiable core for a course in our model we have taken a staged approach. We first compute a minimal set of modules that are in conflict within the given course. Starting from this set we can compute the set of units in conflict based on the units associated to the modules in conflicts.

The algorithm to compute the unsatisfiable core of units can be expressed in B as a recursive function, which in each step removes an element from the input set and then checks the validation predicate, as shown in Figure 4.14. In the figure, variable `units` represents the set of units to be minimized and `acc` is the accumulator for the result. In the function we use several `DEFINITIONS`: *ifte* is a `DEFINITION` that provides an if-then-else construct using lambdas, `CHECK_UNITS` is another `DEFINITION` that checks a set of units for conflicts as described in the previous section and `CHOOSE` is an external function (see Figure 4.14) that implements the mathematical choose operator, introduced to support the translation from TLA$^+$ to B [60]. The recursive B function to compute a minimal set of units minimizes the set `units` of units by stepwise removing units, calling the conflict detection logic and pruning units that have no effect on the outcome of the validation. The result is one of potentially many sets of units that are in conflict, i.e. units that cannot be attended as recommended by the curriculum.

Although ProB can evaluate the function in Figure 4.14 and thus compute its result purely in B we have later replaced it by a so called **external function** for performance reasons. External functions are a mechanism available in ProB to expose functions

```
unsat_core = λ(units, acc).(units <: UNITS \& acc <: UNITS |
  ifte(bool(units = {}), acc,
    ifte(bool(CHECK_UNITS((acc \/ units) \ {CHOOSE(units)})),
      unsat_core(units \ {CHOOSE(units)}, acc \/ {CHOOSE(units)}),
      unsat_core(units \ {CHOOSE(units)}, acc))))
```

```
unsat_core = λ(units, acc).(units <: UNITS & acc <: UNITS |
  IF units = {} THEN
    acc
  ELSE
    IF CHECK_UNITS((acc \/ units) \ {CHOOSE(units)}) THEN
      unsat_core(units \ {CHOOSE(units)}, acc \/ {CHOOSE(units)})
    ELSE
      unsat_core(units \ {CHOOSE(units)}, acc)
    END
  END)
```

Figure 4.14: Simplified version of unsatisfiable core computation in B. The first version uses a `DEFINITION` to represent if-then-else, the second versions uses ProB's if-then-else support in predicates and expressions.

written in Prolog (the implementation language of ProB) to B. This external function `UNSAT_CORE` accepts a predicate, represented as a lambda and a set of input values. The recursive search, implemented in Prolog, stepwise removes an element from the set and evaluates the predicate with the union of the partial result and the remaining set as input. Depending on the result of evaluating the predicate the removed element is either added to the result set or discarded. How the UC of a set of units is computed using the `UNSAT_CORE` external function is shown in Figure 4.15.

```
UNSAT_CORE(λ(units).(units <<: UNITS |
        bool(CHECK_UNITS(units))), units_with_conflict)
```

Figure 4.15: Simplified version of computing the unsatisfiable core of a set of units using an external function.

Having computed an UC of units that lead to a conflict, we additionally compute the set of sessions associated to those units through their groups that are actually in conflict. These are then highlighted in red in the user interface as shown in Figure 4.1. Due to sectioning and multi-session groups, the sessions actually in conflict are often only a small subset of all the sessions associated to the units in conflict. The UC of the sessions is computed similarly to the UC of units. We stepwise remove sessions that do not affect the unsatisfiability of the set of units in the previously computed UC. If a unit is known to have only one session, the session is never removed from the set, as it must be part of the UC.

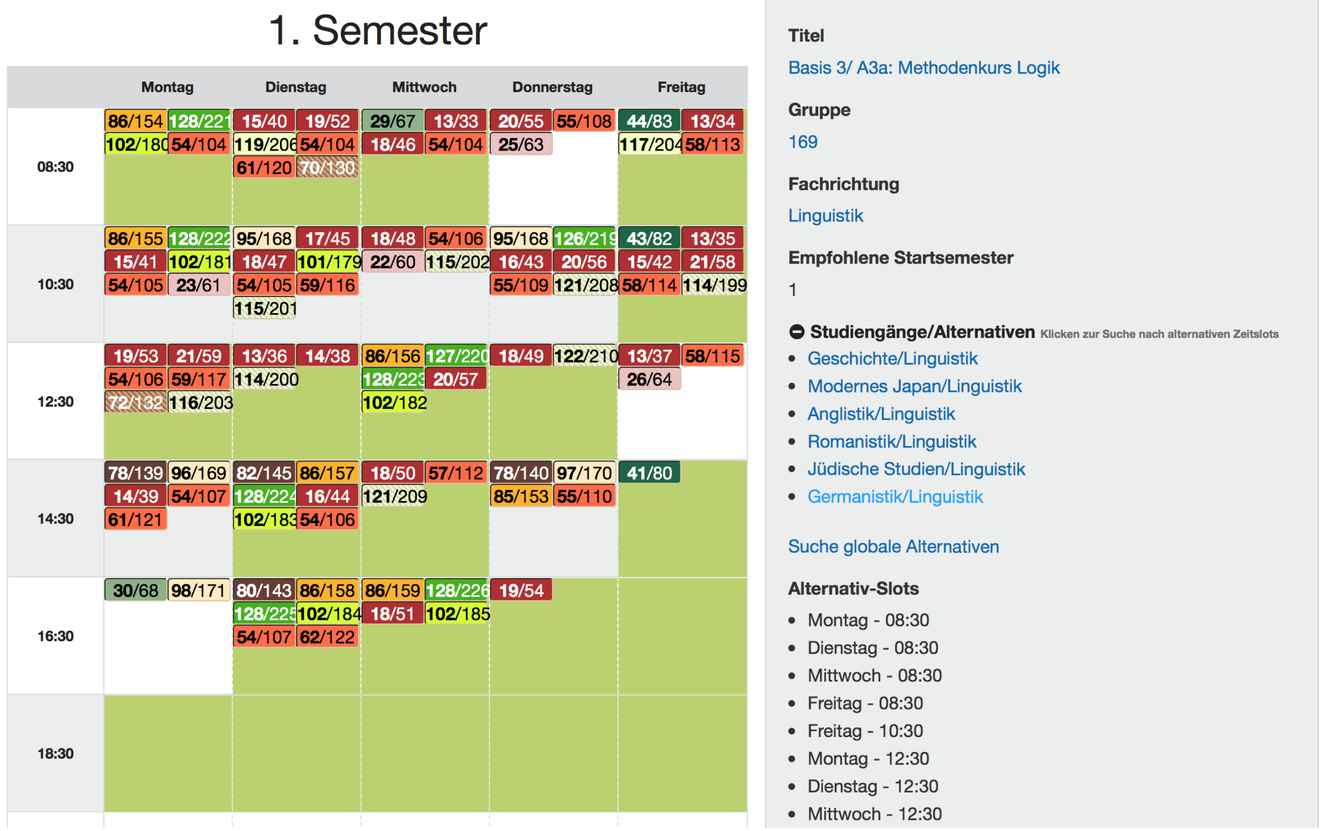
Based on the second version of our model (see Table 4.2 in Section 4.3.7 for details on the different versions) it takes ProB about 18 seconds to compute one unsatisfiable core of modules, units and sessions for each of the 17 infeasible courses in the data set provided by the faculty of *Arts & Humanities* using the implementation based on the external `UNSAT_CORE` function. The same computation using a recursive function written in B takes around 34 seconds to complete.

4.3.5 Improvement – Finding Alternative Time Slots

In case there are feasibility conflicts in a timetable, or just to satisfy changed requirements it is often necessary to move a single session to a different time slot. To avoid creating new conflicts when moving a session to an arbitrary slot we provide a method to compute viable alternatives for a given session, which do not introduce new conflicts. These are computed in the model and if alternative time slots are found these are highlighted in green in the user interface, as shown in Figure 4.16.

The computation is done by first gathering all relevant programs the session is transitively, through its unit, associated to. We then compute all slots, that — when moving the session in question there — satisfy the validation for all relevant curricula at the same time. This approach has the drawback that we compute solutions that only involve doing one change to the curriculum to solve a conflict. This is an area where we want to improve our tool to search for multi-step changes that would solve feasibility conflicts.

Additionally, in certain constellations one change might not be enough to solve the conflicts in all curricula at the same time. Therefore, we additionally allow the users to search for alternatives within one specific program when there are no globally valid alternatives.



1. Semester

	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
08:30	86/154 128/221 102/180 54/104	15/40 19/52 119/206 54/104 61/120 70/130	29/67 13/33 18/46 54/104	20/55 55/108 25/63	44/83 13/34 117/204 58/113
10:30	86/155 128/222 15/41 102/181 54/105 23/61	95/168 17/45 18/47 101/179 54/105 59/116 115/201	18/48 54/106 22/60 115/202	95/168 126/219 16/43 20/56 55/109 121/208	43/82 13/35 15/42 21/58 58/114 114/199
12:30	19/53 21/59 54/106 59/117 72/132 116/203	13/36 14/38 114/200	86/156 127/220 128/223 20/57 102/182	18/49 122/210	13/37 58/115 26/64
14:30	78/139 96/169 14/39 54/107 61/121	82/145 86/157 128/224 16/44 102/183 54/106	18/50 57/112 121/209	78/140 97/170 85/153 55/110	41/80
16:30	30/68 98/171	80/143 86/158 128/225 102/184 54/107 62/122	86/159 128/226 18/51 102/185	19/54	
18:30					

Titel
Basis 3/ A3a: Methodenkurs Logik

Gruppe
169

Fachrichtung
Linguistik

Empfohlene Startsemester
1

Studiengänge/Alternativen Klicken zur Suche nach alternativen Zeitslots
- Geschichte/Linguistik
- Modernes Japan/Linguistik
- Anglistik/Linguistik
- Romanistik/Linguistik
- Jüdische Studien/Linguistik
- Germanistik/Linguistik

Suche globale Alternativen

Alternativ-Slots
- Montag - 08:30
- Dienstag - 08:30
- Mittwoch - 08:30
- Freitag - 08:30
- Freitag - 10:30
- Montag - 12:30
- Dienstag - 12:30
- Mittwoch - 12:30

Figure 4.16: Example of alternatives for the session "Basis 3/ A3a: Methodenkurs Logik". The alternatives in this example are only valid for the course "German studies/Linguistics".

4.3.6 Model Structure

The machine containing the curriculum data is loaded from our main validation machine through a `SEES` clause. The data machine contains the curriculum data as its `PROPERTIES`. In our validation model we derive additional values from these `PROPERTIES`, which are used in the validation predicates. We have represented the validation logic in form of a B predicate that is evaluated in the current state of both machines. The predicate is true if the `CONSTANTS` and state variables represent a valid data set for the curriculum being validated (see also the discussion in Section 4.5).

The validation predicates themselves are structured by means of B `DEFINITIONS`, grouping sub-predicates into `DEFINITIONS`. Using `DEFINITIONS` as the main vehicle of abstraction has some issues which are discussed in Section 5.6.3, but on the other hand there are no idiomatic ways of representing and reusing complex predicates in B.

The externally available features of the machine, basically the API we provide, are exposed as machine operations which are executed from our main application using the animation features of the ProB 2.0 Java API. In the context of each operation we use the different `DEFINITIONS` passing the operation's arguments to the `DEFINITION` and using variables assigned evaluating the `DEFINITION` as result values.

4.3.7 Modelling Stages

In the course of the project, with changing requirements we have stepwise revised our approach, improved our tools and extended our models. Table 4.2 shows the different versions of our models and the key features supported in each version. Although each version of our models builds on the concepts and abstractions introduced in previous versions they required a fundamental change of the underlying data structures and could thus not be mapped to B refinements.

In the first iteration of the project, we started with a data set from only one of the participating faculties, here we solely considered mandatory units. From the raw data we directly generated a B model that was used as input for the validation.

The second iteration was driven by additional data provided by the second participating faculty. In this step we introduced the concepts of mandatory and elective modules as well

Table 4.2: List of key features as supported in each version of our models.

	Version 1	Version 2	Version 3
Courses	●	●	●
Mandatory units	●	●	●
Sectioning (groups)	●	●	●
Modules (mandatory and elective)		●	●
Elective units		●	●
Abstract units			●
Levels with arbitrary nesting			●

as elective units. Data was provided in incompatible formats, leading us to create tools to import both formats into a common B representation instead of directly generating a B model. Later we introduced an intermediate storage to separate the handling of the different input formats from the generation of B machines for the data. The number of entities used in the validation with this version of our models is shown in Table 4.3. The B machines generated for the data contained 2247 lines for the AH data set and 1724 lines for the BAE data set. In this version there were faculty specific machines that modelled the curricula and validation rules that are distinct for each faculty consisting of 377 lines for the AH and 734 lines for the BAE data sets. Common aspects and rules were shared between the models (301 lines in total), e.g. the rules on how to compare and validate sets of teaching units.

In the third iteration we added the computer science (**CS**) curricula to our project (consisting of computer science as a major and all available minors). At this point we added the concept of abstract units as additional indirection and have moved to a common representation for the raw data in all faculties. We have abstracted the particularities in faculty's curricula, such that we can model all of them in one B model (708 lines of B in total) that can be used to validate all curricula we have collected. The size of the data collected in this iteration is outlined in Table 4.4.

Table 4.3: Number of entities for the faculties of *Arts & Humanities* (AH) and *Business Administration & Economics* (BAE) in the second version of our model.

Data Set	Courses	Modules	Units	Sessions
AH	67	1	128	249
BAE	6	84	221	332

Table 4.4: Number of entities for the faculties of *Arts & Humanities* (AH), *Business Administration & Economics* (BAE) and the *Computer Science* (CS) curricula in the third version of our model.

Data Set	Courses	Modules	Abstract Units	Units	Sessions
AH	43	304	606	1390	1811
BAE	26	183	300	277	406
CS	12	98	122	148	385

4.4 From Model to Application

The goal of our project is to create an application that can be used by those responsible for planning the curricula at the different faculties participating in this project. In this sense we want to create an interactive application that is simple to use and allows its users to interact with the previously described features provided by our modelling.

4.4.1 System Architecture

To this end we have created a multi-tiered application; composed of the **model** as described in the previous sections, a **storage** layer composed of the data provided by faculties, tools to import this data into a database and tools to generate a B model from such a database. As a browser is currently one of the most widely available platforms, we have chosen to implement the **user interface** as a rich internet application. Finally, we have the **core** application or **server** layer, this layer embeds the formal model and ProB, mainly providing two things: a simple GUI to load a database and start the application and second a REST API that manages the interaction between the **presentation** layer and the **data** and **model** layers. All requests from the **user interface** to the **server** are made as HTTP requests sending and receiving JavaScript Object Notation (JSON) formatted data. The architecture is visualized in Figure 4.17.

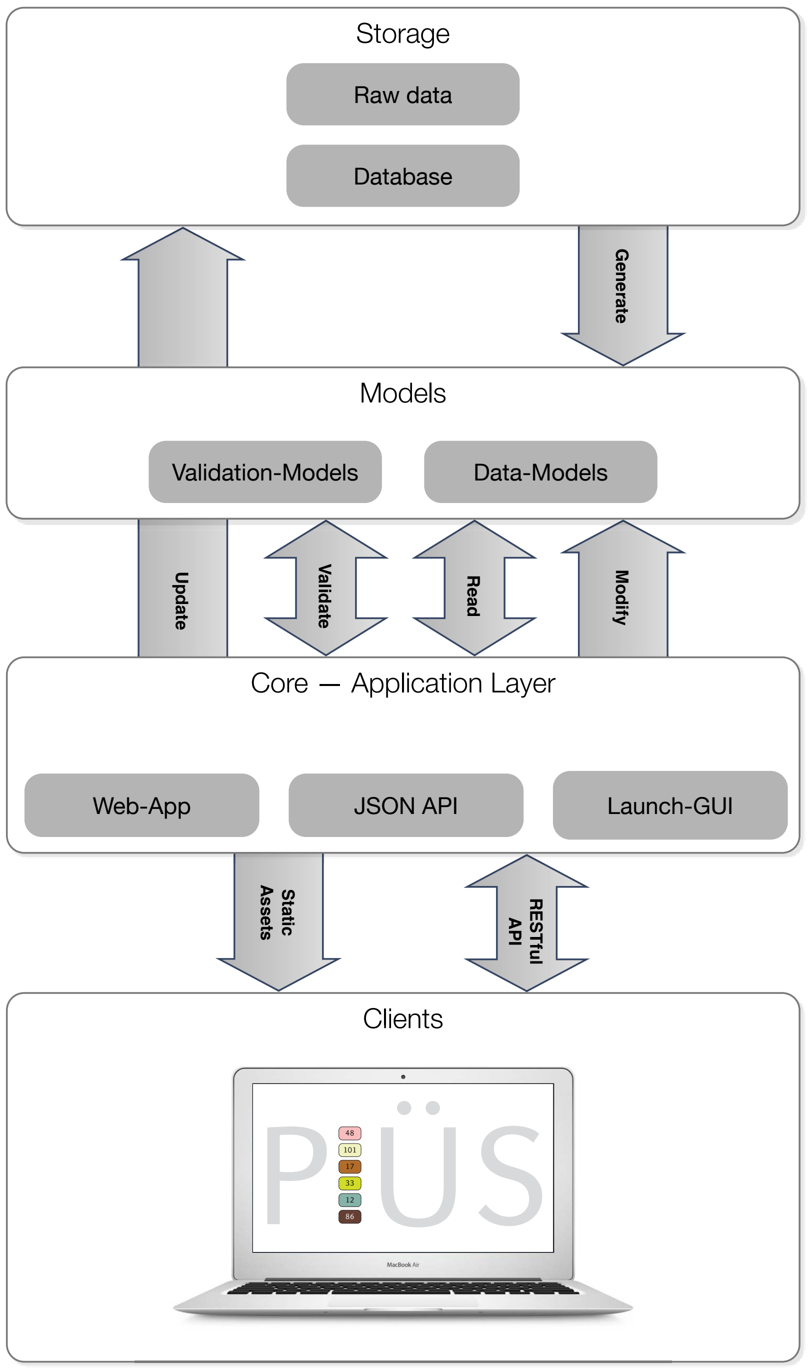


Figure 4.17: Schematic representation of the system architecture, showing the different components and the interaction between them.

```
...
public def checkFeasibility(String... courses) {
    ...
    def op = "check"
    def names = courses.collect{'"' + it + '"'}.join(", ")
    def predicate = "courses = {${names}}"
    executeOperation(op, predicate)
}

protected def executeOperation(String op, String predicate) {
    if (trace.canExecuteEvent(op, predicate)) {
        trace = trace.execute(op, predicate)
        def trans = trace.getCurrentTransition()
        def return_values =
            trans.evaluate(FormulaExpand.expand).getReturnValues()
        ...
        return return_values.collect { Translator.translate(it) }
    }
    false
}
```

Figure 4.18: Interface between the application and the formal model to execute an operation using the ProB Java API.

The database generated from the raw data serves two purposes, first it is used to generate the input for the validation models and second it is used to populate the user interface of the application in the client. To bootstrap the application the user needs to select a database from which we generate the B representation of the curriculum data. The application launches the GUI in the browser once it has initialized. When accessing the application the browser loads the static assets to run the application from the **core**. When the user interface is initialized, it requests the curriculum data from the **core** using the RESTful API. E.g. to load and present the session data the client requests the data by invoking the `/api/sessions` endpoint via a `GET` request. The **server** responds with a collection of sessions represented as JSON objects, a simplified session in such a response is shown in Figure 4.19.

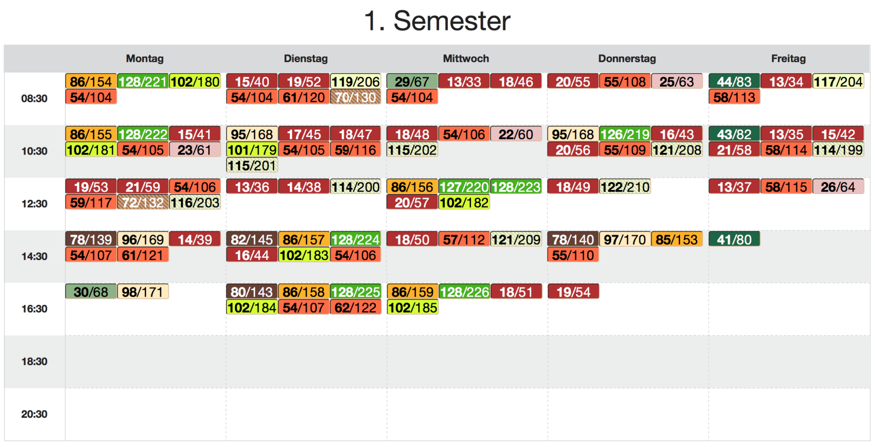
The validation models are exported from the application bundle to the same location as the generated data machine and loaded with ProB. ProB is embedded in the application using the ProB 2.0 Java API, which exposes the available features to applications running on the JVM. We interact with the model using the animation features of ProB. This allows us to execute one machine operation at a time and use the computed results in our application. We have created a simple abstraction that exposes the features of our model and invokes the corresponding operations on the validation machine, as shown in Figure 4.18, each operation is executed by calling the `execute` method on a `trace` object. A `trace` represents a specific order of executed operations and the current state (variables) of an animation. `execute` is called with the name of the operation to be executed and a predicate describing the values for the parameters of the operation, it evalutes the corresponding machine operation and returns a `trace` object representing the new state. In the example the variable *courses* should be equal to the set of names passed to `checkFeasibility`. Calling the `execute` method on a `trace` object performs one animation step, i.e. it computes one state transition, and returns a `trace` object representing the newly current state of the animation.

When interacting with the application the user can trigger different actions that invoke specific HTTP endpoints on the **server**. Actions might involve the validation of a specific curriculum, generating a PDF to distribute for validated curricula, detecting conflict sources, etc. For example if the user triggers the validation of a specific curriculum in the user interface the **presentation** layer will issue an HTTP call to a specific endpoint in the **server**. The **server** validates the requests and if the request is valid evaluates the corresponding operation in the validation machine using the course name provided as part of the HTTP request as a parameter for the machine operation. The result computed by ProB for the operation, in this case the result is the set of chosen units, groups and semesters, is translated from the B representation to a corresponding JSON representation and sent to the client to update the user interface.

The representation of the sessions in the corresponding time slots and semesters in the user interface, shown in Figure 4.20, is inspired by the analogue timetable planning boards that are used in schools. Every single session is represented by a simple box that carries identifying information, e.g. we are using colours, to distinguish the department and each box has a symbol, that shows which unit the session belongs to. Boxes can be dragged to different time slots to update the timetable. Additionally, each box can be greyed out or highlighted to visualize further information such as conflicts or exclusions.

```
{
  "sessions" : [
    {"id" : 1, "time" : 1, "day" : "tue", "group" : 1, "unit" : 1,
        "courses" : [1],
     "allSemesters" : [1,3,5], "abstract_units" : [1]}
  ],
  "abstract_units" : [
    {"id" : 1, "title" : "Introduction to Formal Methods", "key" :
        "P-CS-L-INFO1A"}
  ],
  "courses" : [
    {"id" : 1, "longName" : "BSc Computer Science", "key" :
        "BK-CS-H-2013", "shortName" : "cs"}
  ],
  "units" : [
    {"id" : 1, "title" : "Introduction to the B Method", "key" :
        "110000"}
  ]
}
```

Figure 4.19: Simplified server response from `GET /api/sessions` when loading all available sessions.


1. Semester

	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
08:30	86/154 128/221 102/180 54/104	15/40 19/52 119/206 54/104 61/120 70/130	29/67 13/33 18/46 54/104	20/55 55/108 25/63	44/83 13/34 117/204 58/113
10:30	86/155 128/222 15/41 102/181 54/105 23/61	95/168 17/45 18/47 101/179 54/105 59/116 115/201	18/48 54/106 22/60 115/202	95/168 126/219 16/43 20/56 55/109 121/208	43/82 13/35 15/42 21/58 58/114 114/199
12:30	19/53 21/59 54/106 59/117 72/132 116/203	13/36 14/38 114/200	86/156 127/220 128/223 20/57 102/182	18/49 122/210	13/37 58/115 26/64
14:30	78/139 96/169 14/39 54/107 61/121	82/145 86/157 128/224 16/44 102/183 54/106	18/50 57/112 121/209	78/140 97/170 85/153 55/110	41/80
16:30	30/68 98/171	80/143 86/158 128/225 102/184 54/107 62/122	86/159 128/226 18/51 102/185	19/54	
18:30					
20:30					

Figure 4.20: Screenshot of the timetable for the first semester including all available sessions.

4.5 Related Work and Discussion

Time-Tabling Automatic timetabling is a long-standing area of research [57], so more work than can be covered here has been done on this topic. This area has seen interest from different research communities, as it presents a challenging problem with real world applications for a variety of approaches. There is research based on metaheuristics and genetic algorithms, that aim to improve timetables through mutation [36, 94]. There is also research on this problem using SAT techniques, such as the work done by Asín Achá and Nieuwenhuis on timetabling with SAT and MaxSAT [5], SMT solving using Z3 [41], Answer Set Programming [11], Integer Linear Programming [38, 49, 112] and based on constraints [43, 103, 111]. In 2002, 2007 [44] and 2011 [109] the International Timetabling Competition (ITC) took place, which provided a set of benchmarks to drive and compare research [82]. Many tools, commercial and research focused, have been created in the area of timetable generation, among them Yeung [130] built a tool based on Kodkod [125], the relational model finder used by the Alloy Analyzer, to generate course schedules for students based on global constraints and individual constraints provided by the students. UniTime[2] is a scheduling system for courses and exams based on a constraint solving system created by Müller [102].

Could other approaches have been used?

Model Checking instead of constraint solving In principle one could also have encoded our validation problem as a model checking task, i.e., encoding the choice of units as B operations and asking a model checker whether a successful sequence of choices exist.This, however, would have been less tractable, as the model checker evaluates each B operation in isolation, without regard for the overall goal of finding a conflict-free path through the curriculum. Model checking thus amounts to naive enumeration. A real constraint solver, on the other hand, may be able to detect very early that a certain partial combination of choices will never lead to a successful outcome.

This point has also been discussed in [90]. To illustrate this point recall the n-Queens puzzle from Chapter 3, which is a simple form of time-tabling: placing n queens on an $n \times n$ chessboard so that no queen attacks (aka conflicts) with any other queen.

[2]See `http://www.unitime.org` - [Online; accessed 31-March-2017]

ProB can solve this constraint in about half a second for $n = 70$ (on an Intel Core i5 CPU running at 2.67 GHz using ProB 1.6.2-beta1 (revision: eec70f07)), as shown in Table 3.1. Solving an encoding of the n-Queens puzzle as a model checking task of this size is infeasible using explicit state model checking. E.g., using TLC [131] it takes more than half an hour to solve a TLA^+ encoding of n-Queens for $n = 14$; one issue being the breadth-first strategy of TLC, which basically means that all solutions are found more or less at the same time. Using Spin 6.4.5 [67] and a depth-first strategy, a Promela encoding [15, Listing 11.5][3] of the problem can be solved in 0.09 seconds for $n = 14$. However, for $n = 28$ Spin also reaches its limits, taking already about half an hour to solve a Promela encoding of the n-Queens puzzle. ProB can solve n-Queens in 0.01 seconds for $n = 14$, 0.05 seconds for $n = 28$.

Other tools or languages for constraint solving Our formal B model could probably just as well have been expressed in another state-based formal method; we return to this issue in the conclusion. The constraint solving capabilities are crucial for our application. Hence, tools like the model checker TLC [131] or the animators coreASM [48] or AnimB[4] cannot be used for (variations of) our present model.

ProB relies on constraint logic programming [74]. Other successful approaches to constraint solving in the context of formal methods are SAT and SMT solving [45]. Indeed, an alternate constraint solving backend for ProB, which uses the Kodkod library [125], translating first-order relational logic into SAT problems has been created [108]. Unfortunately, we were unable to use this backend here, due to fundamental performance issues for relations over large domains.[5]

Another promising technology is SMT, where one can circumvent the above SAT issue of dealing with large domains by using theories. A translation of Event-B formulas into SMT-LIB format is available [40], and has proven very successful for proof. For constraint solving (aka model finding) the issue is somewhat different [108]. For example, even

[3]Manually adapted for various values of n. With supertrace/bitstate hashing the solution is not found.
[4]`http://wiki.event-b.org/index.php/AnimB` - [Online; accessed 31-March-2017]
[5]In our experiments, Kodkod was either orders of magnitude slower at various tasks (such as determining programs with units in common), or was unable to achieve the SAT translation (`CapacityExceededException`).

the simple n-Queens problem for n=4 cannot be solved by current SMT solvers such as Z3 [39] or CVC4 on the translation of the SMT-Solver integration for Rodin [40].[6]

In conclusion, while ProB's constraint solving based on constraint logic programming has some drawbacks over SAT or SMT based approaches (no learning for example), its ability to deal well with large relations, integer values and symbolically with infinite or recursive functions make it well suited for the time-tabling application described in this chapter. We will compare B and ProB with regard to this problem with alternative approaches in Chapter 6.

4.6 Future Work and Conclusion

In this chapter we have presented the application of formal methods to a novel domain. We have successfully modelled university curricula and timetable validation in B in a way that captures the domain constraints and can be executed using ProB.

Our models scale well within the scope of real-world data we have used in our project, e.g., we are able to validate the timetables for *all* the programs (based on the second version of our models, see Table 4.2) offered by the faculty of *Arts & Humanities* in 4 seconds and to compute a minimal unsatisfiable core of sessions for each of the 17 infeasible programs in 18 seconds in total. In Chapter 6 we will evaluate the performance of ProB in more detail.

The use of a high-level language to model this problem allowed us to decouple the model from the solving strategy and thus permitted us to easily evolve the models during the process of capturing all the domain information and requirements.

Modelling this problem and creating the application on top of it has served as a driver for ProB and its related tools. It has been useful to uncover bugs and performance problems and the ongoing project will contribute to evolve ProB and possibly also the B language itself.[7]

[6] A fundamental issue seems to be that the current SMT-LIB translation sometimes encodes finite B relations and B sets as infinite functions. boolean values.

[7] For example, we have already added if-then-else and let-constructs for expressions.

We are aware that a high-level model of a constraint problem cannot compete with either low-level solutions or dedicated solvers specialized for this class of scheduling problems. We want to improve the capabilities of ProB in this regard and in Chapter 6 we perform an extensive evaluation and compare our approach to other solutions written directly in more tractable formal methods, such as Alloy [72], SAT and SMT encodings, as well as a lower-level Prolog encoding using clp(FD) [27]. We will evaluate the performance aspects, but also compare the complexity of the models, the complexity of validating and modifying the models, and the ease of embedding the model in a production system. Indeed, we believe that developing and adapting a high-level non-algorithmic model is considerably easier, and that formal method tooling can help in validating the model. Our goal is to move formal models from design documents to artefacts embedded in running systems. We are still far away from truly "executable mathematics", but in this chapter we have shown that formal model-based problem solving is starting to become practically feasible.

Part III

Evaluation and Outlook

5 Evaluation of the Software Solution

5.1 Introduction

This chapter is based on an article titled "Using B and ProB for Data Validation Projects" [61] coauthored with Dominik Hansen and Michael Leuschel presented at the ABZ 2016 conference in Linz, Austria. The work on the validation of railway topologies introduced below was conducted independently by Dominik Hansen.

We have so far argued that constraint satisfaction and data validation problems can be expressed very elegantly in state-based formal methods such as B. In this chapter we introduce a second independently developed project about the validation of railway topologies. Both projects are based on the B language and use ProB as the central validation tool.

Combining the experiences from the case study presented in the previous chapter and this second project, we present a general structure of a data validation project in B and outline common challenges along with various solutions. We also discuss possible evolutions of the B language to make it (even) more suitable for such projects.

5.2 Background

Data validation[1] ensures that software operates on correct, clean data and is typically done by checking validation rules or constraints. We have previously argued that B is a very expressive language to encode constraint satisfaction problems, and many data

[1] `http://www.data-validation.fr` - [Online; accessed 31-March-2017]

validation problems can be expressed as such. Other works have demonstrated that B is useful to express properties about data and to validate them using ProB, particularly in the railway domain [2, 7, 8, 9, 84, 90].

We have used the B language to express parts of our program's domain logic and the rules to validate data, and embedded these B models into running applications by executing the formal models with ProB without relying on code generation. It would also be possible to express these kinds of validation problems in other formal languages such as Alloy [72] and TLA$^+$ [83]. Based on our experiences with these languages and the corresponding tools, we believe that the combination of B and ProB best meets the requirements for the data validation task. Our explicit goal is to explore the applicability and scalability of this combination for projects of industrial strengths.

Based on two projects, the case study from Chapter 4 and the one described below, we will discuss different aspects of using B within such an application and discuss the approaches taken as well as the limitations encountered, i.e. where we had to depart from or extend the language to suit our needs.

Validation of railway topologies is the independent project discussed in this chapter and part of a collaborative research project with Thales Transportation Systems GmbH on applying formal methods for the software development process of the Radio Block Centre (RBC). The RBC is a communication unit of the European Train Control System (ETCS) exchanging messages with trains and interlockings. One of our challenges in this context is to validate the so-called engineering rules over concrete track data. The track data is a representation of the real railway infrastructure and signalling system. Engineering rules are implementation-related rules which result from the concrete RBC implementation. This means, that the concrete RBC implementation is guaranteed to work correctly only if the concrete track data satisfy the engineering rules. For example, a simplified engineering rule requires that two signals for the same direction should not be located at the same position. The modelled engineering rules are validated on different track topologies. The biggest topology contains 1362 track segments, 457 points, 1089 balise groups and 445 signals.

```
!signal1, signal2.(
  signal1 : Signals & signal2 : Signals & signal1 /= signal2
  & Signal_Direction(signal1) = Signal_Direction(signal2)
   ⇒ not(Signal_TrackSegment(signal1) = Signal_TrackSegment(signal2)
         & Signal_Position(signal1) = Signal_Position(signal2)))
```

Figure 5.1: Modelling of an engineering rule as a validation predicate.

As discussed in Section 2.1.2 the idea of using formal method languages and tools to perform data validation has been explored in the past, e.g. by Abo and Voisin [2] or Lecomte et al. [84] among others. Our intention in this chapter is to outline the common structure and challenging aspects of data validation projects based on what we have identified in both projects. The domains and requirements of these two projects are quite different, and all work has been done independently (i.e. by different people). Still, similar challenges were faced during the modelling process. Both projects rely on ProB as the tool to evaluate the models.

In the following sections we will name these challenges, discuss different language constructs of B and argue how they can be applied in modelling data validation problems. Moreover, we will outline areas where we have extended the B language to overcome some limitations we faced evaluating the models with ProB.

5.3 The Big Picture

Before describing the details of the data validation process we will discuss the big picture, outlining the design and architecture that emerged from both projects mentioned in the previous section.

The general idea is to create B models that define validation predicates which are evaluated against the state of the model. The variables and constants are derived from external data we want to validate. Figure 5.1 shows the formalization of the validation rule mentioned in the introductory section where two signals should not be located at same position if they are valid for the same direction.

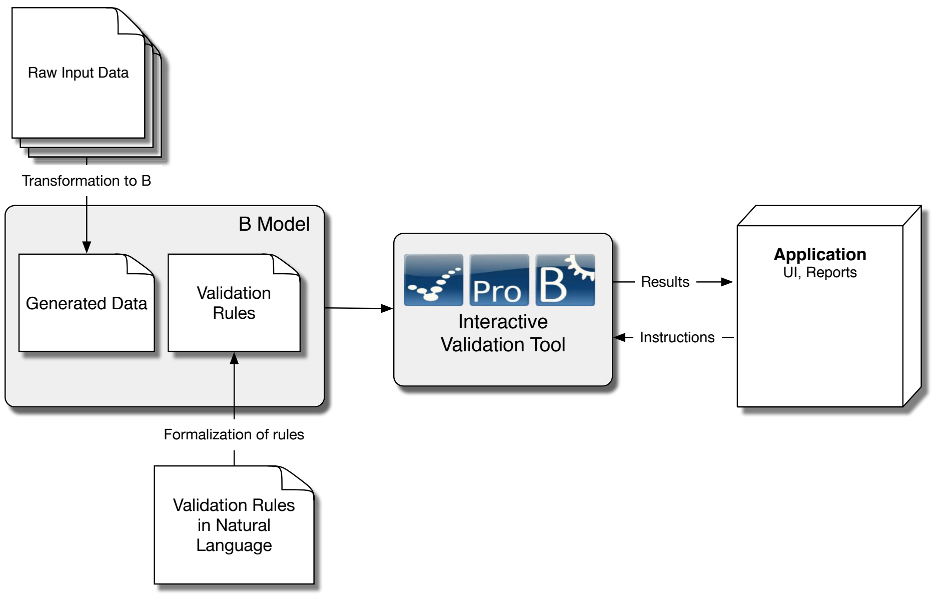


Figure 5.2: Generalized architecture of ProB based data validation project

The projects discussed in this chapter follow the general architecture shown in Figure 5.2. By building data validations tools based on the B language we have identified the following concerns: The first is getting the external data from a given source into a B model which is discussed in Section 5.4. Choosing a way to represent the data is a further concern, where it is important to choose a representation and B data types suited for the validation process while keeping the import process as simple as possible; this is discussed in Section 5.5. Some validation rules rely on derived data (e.g., signals reachable from a point) which has to be computed from the imported data. In Section 5.6 we present different approaches to structure derived data in B. One purpose of the B Method is to model algorithms and prove their correctness. However, are these models suitable for use by ProB to calculate results? Section 5.7 describes different approaches to model an algorithm in B such that it can be efficiently evaluated by ProB. Another concern is how to control the validation process from an external application. In Section 5.8 we discuss different ways to interact with the model. Finally, in Section 5.9 we briefly discuss how to reuse an existing validation model in similar projects.

5.4 Preparing Data for Use With a B Model

When used for data validation, our tools obviously depend on externally provided data [2, 87], which has to be converted to B format in order to be validated with ProB. Raw input data is provided in a variety of formats as used in the different domains such as Excel, CSV or XML documents.

In both projects we have opted to create tools that read and parse the externally provided data and generate a text file containing a B model of the data. The data will be accessible as a series of constants in the model.

Having an external tool keeps any knowledge about the raw data format out of the B models; but of course it raises a series of concerns. One is having to maintain an additional tool which has to generate valid B. Also the chosen data representation has to be kept in sync between the import and the validation tools.

Another concern is that, in a safety critical environment, the import tool itself has to be validated. The topology validation project takes a direct approach by avoiding putting too much knowledge into the transformation step, keeping it as simple as possible. In this approach the transformation process maps the input structure of the data (XML) to B data structures and copies the values of attributes as uninterpreted strings. To ensure that all data from the input document is represented in the B model we use a back-translation (from the B model to XML) and compare the generated XML document with the source document. The back-translation is done in order to certify the translation tool and ensure that no data has been left out.

As discussed before, in the case of the curriculum validation tool the data is not only used for validation purposes but also to populate the application's user interface, hence we have chosen a two step approach that does not directly generate a B machine, but rather imports the data into a database. The information in the database is later used to generate the actual B representation of the model at runtime. Additionally, the database is used in the application to persist changes and as the data source for the user interface. Since the data is used in multiple places we map the values in the raw data to the most adequate types in the database and later to the corresponding B types.

Are there any alternatives? There are many alternative approaches that could be pursued to import data into a B model. E.g. instead of generating a B model with the data as constants, it would be possible to have B operations which incrementally add values to variables containing the data. These operations could be executed in various ways, e.g., using the Java API for ProB. Finally, ProB exposes external functions to B that make it possible to, e.g., load data from CSV files; these features could be extended for additional data sources (see Section 5.7.3).

5.5 Data Representation

Hand in hand with the decision on how to import data into a B model goes the choice of proper B data-structures to represent the data. This representation should ideally follow the structure of the source data, and additionally lend itself to be used and manipulated in B. Choosing a good representation for the problem is crucial for the complexity and readability of the model. In "Understanding the differences between VDM and Z" [62] Hayes et al. discuss some of these issues on the examples of a simple database in VDM and Z. In B, one could encode database records as nested pairs. A quaternary relation over course identifiers, semester, weekday, and starting hour could thus be represented as:

```
db = { (((course1 ↦ sem2) ↦ monday) ↦ 14),
       (((course2 ↦ sem1) ↦ friday) ↦ 9) }
```

In order to access the first and second element of a pair, B provides the prj_1 and prj_2 operators. However, in B accessing a certain field of a nested pair is very cumbersome, as we have to unfold the nested pair until we reach the desired field.[2]

Another alternative is to use records with named fields:

```
db = { rec(course_id: course1, semester: sem2,
           weekday: monday, starting_hour: 14),
       rec(course_id: course2, semester: sem1,
           weekday: friday, starting_hour: 9) }
```

[2]In addition, the types of the arguments have to be provided for prj_1 and prj_2; e.g., $\mathrm{prj}_2((COURSE \times SEMESTER) \times WEEKDAY, \mathbb{Z})(v)$.

We can easily access a field of a record r by using the quote operator: `r'course_id`. Compared to the encoding as nested pairs, records are more readable, especially if there are a large number of fields. Otherwise, constructing a record is more verbose than constructing nested pairs. Since this part of the model is automatically generated, the verbose encoding is not an issue.

A third alternative is to create B functions for each attribute of the data record mapping a unique identifier to the corresponding attribute value. An identifier of a data record could be a unique number generated by the translator or a certain attribute of the data record. In case of our example, we could choose the attribute `course_id` as the unique identifier:

```
course_id__semester = {course1 ↦ sem2, course2 ↦ sem1}
course_id__weekday = {course1 ↦ monday, course2 ↦ friday}
course_id__start_hour = {course1 ↦ 14, course2 ↦ 9}
```

While this approach works well for simple tables such as in Excel or CSV documents, it would become inconvenient for nested data structures, e.g. if a value of a field is itself a set of data records such as a sub-tag of an XML document. In this case, the translation tool first has to transform the nested data structure to a relational database schema. Subsequently, the translator has to create a B function for each attribute of each table of the relational database.

One advantage of the last alternative is the handling of optional fields. Indeed, when no field value is present for a data record, we just omit the corresponding identifier from the domain of the accessor function for the corresponding field (i.e., we use partial functions rather than total functions). For the other two approaches optional fields pose more of a challenge. Due to the strong and strict typing of B it is not possible to create partial records or to omit a field of a nested pair. One solution is to introduce a special `NULL` value for each B datatype, e.g. the empty string (`""`) for the `STRING` type. However, we have to ensure that the `NULL` value is not a regular value in the source data. For other data types it is more intricate, e.g. which number to choose for `INTEGER` typed values or how to represent this at all for `BOOL` typed values.

This directly leads to a further aspect of the translation. How should values in data records be represented? They could be represented either as uninterpreted strings of data copied verbatim from the raw data input in the transformation step. Alternatively the data values could be represented using the most appropriate B data types, e.g. `INTEGER` for numbers, and enumerated sets for values from a set of known values. The first approach has the advantage of a very simple translation process and that all relevant knowledge about the data is encoded in the B model. The drawback is now, however, that the data has to be translated in the B model, which typically requires extensions to the B language which are available in ProB (e.g. transforming a `STRING` value to an `INTEGER` value).

In both projects, we have chosen the record representation for the data. As already mentioned in the previous section, the timetabling tool maps the raw data to the corresponding B data types. In case of the topology validation project, all data values are represented as uninterpreted strings and the processing of these strings is part of the B model.

5.6 Means of Abstraction – Structuring and Auxiliary Constructs

In general, abstractions [1] in programs and also models control complexity, encourage reuse and make testing easier. Different parts of the B language offer different ways to abstract and structure models and programs. There are certain concepts that are applicable at the machine and operation level while others are applicable on the predicate and expression level.

5.6.1 Machines and Operations

On the machine level sub-problems can be structured as machines for each sub-aspect which communicate through the execution of operations. The visibility of machines and their variables and operations can be controlled using different machine composition mechanisms such as `SEES`, `USES` and `INSTANCE`.

```
CONSTANTS
  ConflictRelation
PROPERTIES
  ConflictRelation =
    UNION(r1,r2).(r1 : SignalRecords & r2 : SignalRecords
      & r1'elementID /= r2'elementID
      & r1'trackSegment = r2'trackSegment
      & r1'position = r2'position & r1'direction = r2'direction
      | {r1'elementID ↦ r2'elementID})
```

Figure 5.3: Calculating the conflict relation of two signals placed at the same position.

On the level of a single operation the substitution language provides several expressions that are useful, either if-then-else for control flow or let constructs to introduce scoped variables.

5.6.2 Expressions and Predicates

Within the mathematical language of B, constants can be used to globally save precomputed values whose computation might be expensive and should not be evaluated more than once. Figure 5.3 shows the calculation of the conflict relations of two different signals placed at the same position and valid for the same direction. Note, that the calculation corresponds to a SQL statement making a self join on a signal table. Moreover, constants can be used to store certain calculations in the form of lambda functions which can be used in different parts of the model. However, constants are not applicable for intermediate results which can not be precomputed globally because they depend on additional information or parameter values.

Language Extensions for Predicates and Expressions

We have extended the predicate and expression subset of B supported by ProB to make it easier to write complex validation rules. We have included two constructs from the substitution part of the language that proved to be useful when creating validation rules.

IF-THEN-ELSE It is possible to express a condition in B using a union of lambdas and function application, in order to mimic and if-then-else construct. Nonetheless the evaluation is eager and the use of it a bit cumbersome. For this reason we have made the IF-THEN-ELSE substitution syntax available in expressions and predicates. Figure 4.14 shows an example of using both methods of expressing conditional execution for computing an unsatisfiable core.

`LET` for Predicates and Expressions. In complex expressions or predicates it is often useful to introduce local variables as a shorthand for certain values or expressions. B only supports `LET` in the context of substitutions, nonetheless it might be useful for predicates and expressions. In plain B there are several ways to achieve this: For example, an existential quantification (`#x.(x=E & P)`) can be used within predicates to achieve a result similar to a `LET` ProB tries to identify existential quantifications that only have a single value and treats them specially. Within set-comprehensions an existential quantification could also be used (`{x| #y.(y=E & P)}`), but the following pattern using the domain of a set of pairs is (generally) more efficient in ProB: `dom({x,y| y=E & P})`. For expressions which denote a set of values, one can use `UNION(y).(y=E| S)`.

To avoid using these workaround that rely on the detection of patterns in ProB we have added support for the `LET` constructs from the substitution syntax to the B language for expressions and predicates. See Figure 5.4 for an example.

5.6.3 DEFINITIONS

Besides lambdas, one of the available methods of decomposing larger predicates or expressions into smaller reusable components are `DEFINITIONS` (comparable to macros). `DEFINITIONS` are textual replacement constructs similar to macros, that can accept parameters which are replaced verbatim in the body of the `DEFINITION`. They can be nested arbitrarily as long as the resulting expression or predicate is syntactically correct. The use of `DEFINITIONS` carries some issues that have to be kept in mind. Although they are textual replacements, ProB requires every definition to be syntactically correct on its own, so certain compositions patterns are not possible.

```
DEFINITIONS
EXAMPLE(aa, bb) == LET
    va, vb
  BE
    va = aa
    &
    vb = bb
  IN
    <predicate over va and vb>
  END;
```

Figure 5.4: Using a `LET` inside a definition for scoping.

Care is needed with regard to naming conflicts, quantifications not captured in the `DEFINITION` and issues such as unintentionally performing repeated computations. Unintentional repeated computations might happen when writing a complex expression as a parameter to a `DEFINITION`, where in case of a function or lambda the computed value is passed to the function as argument in the case of `DEFINITIONS`, the expression is inserted verbatim at every location the variable appears, causing the expression to be repeatedly evaluated at each location. The same care as with repeated computations is needed in the case of naming conflicts, given that `DEFINITIONS` do not provide any means of scoping, using a `DEFINITION` in the wrong context might lead to unexpected naming conflicts and type errors (see, e.g., “A Theory of Hygienic Macros” [63]).

Take for example the definition `even(x) == (#y.(y:1..x & 2*y=x))`. Evaluating the predicate `even(4)` yields true. However, if we have a machine variable `y` whose value is 4 and evaluate `even(y)` we obtain false, because the definition call was rewritten to `#y.(y:1..y & 2*y=y)`.

Regarding unintended repeated computation of arguments, the arguments of a definition may get replaced multiple times and then also executed multiple times by ProB. Take, e.g., the definition `POW3(x)==x*x*x` and the call `POW3(f(1))`. The latter gets transformed into `f(1)*f(1)*f(1)`, resulting in repeated computations of `f(1)`. A pattern we have used to avoid this is to create a variable within each `DEFINITION`, which is assigned with the passed argument and used instead of the original parameter to avoid unintentionally causing repeated computations of the same expression as shown in Figure 5.4.

For the reasons described above, `DEFINITIONS`, although they are a useful method to store and structure expressions and predicates, should be used carefully, in particular for big expressions with parameters.

A concept that could present a useful extension to B is that of `predicates` as present in Alloy [72]. Alloy distinguishes between named expressions and named predicates. Predicates work as functions, providing proper scoping. Additionally, the parameters are bound and evaluated in the context the predicate is used avoiding repeated computations due to text replacement. Both characteristics make it possible to safely, in distinction to `DEFINITIONS`, reuse and combine predicates.

5.7 Using B to Express Computations

Not every validation rule can be easily described using B predicates. In those cases it can be more convenient to describe these concepts using recursive rules or as fixpoints of iterative algorithms. In this section we show how this can be achieved in a natural B style, while also ensuring that the resulting algorithms can be executed efficiently.

We will discuss different approaches to model an algorithm for sorting a set of numbers into an ordered sequence. Note that the B Method does not provide a built-in operator to sort a set. In the actual applications we used these techniques for more complicated constructs, such as a search on a rail way topology with various termination conditions.

5.7.1 Machines and Operations

First we will discuss the approach of using machines and operations to express the required functionality. Using the machine and substitution semantics of B to express computations has the clear advantage of having all tools and features of the B Method at our disposal. Figure 5.5 shows a stateless query operation calculating the sorted sequence for a given input set.

However, ProB is not able to evaluate the operation efficiently, i.e. it does not scale for large input sets. Indeed, a naive execution of `Sort_OP` would calculate all possible permutations of the input set to then reject all but one, which is the sorted sequence.

```
out_sortedSequence <-- Sort_OP(p_set) =
  PRE p_set : POW(INTEGER) THEN
    out_sortedSequence : (
        out_sortedSequence : iseq(p_set)
        & ran(out_sortedSequence) = p_set
        & !i.(i : 1 .. size(out_sortedSequence) - 1
              ⇒ out_sortedSequence(i) < out_sortedSequence(i + 1)))
  END
```

Figure 5.5: Query operation

ProB's constraint solving can overcome this exponential complexity to some extent,[3] but for larger sequences we are a far cry from the performance of ordinary sorting algorithms. Following the refinement principles of the B Method we can implement the abstract operation by a concrete sorting algorithm. Figure 5.6 shows a selection sort (MinSort) implementation in B. The operation `Sort_OP` exposes the algorithm as a single operation which can be used several times and embedded in different machines.

ProB provides various optimizations for while loops. First, an interesting point is that the variant is evaluated upon entry and gives ProB an upper-bound on the number of iterations.[4] If a certain threshold is exceeded, ProB will pre-compile the body of a while loop, by precomputing all parts which do not depend on variables modified in the loop.

In our approach, we are not interested in proving the concrete algorithm to be a correct refinement of the abstraction. However, we are interested in the correctness of the sort implementation. Therefore, we use the predicate of the abstract operation as an invariant respectively an assertion on the output of the concrete operation. Note, that in this case ProB is able to check that the predicate holds for a concrete value even for a large input set. Moreover, the termination of the sort algorithm is ensured using a loop variant which is observed by ProB. For more complex algorithms such as different search

[3]ProB can compute `Sort_OP({3,55,22,44,1,100,20,40,55,88,10,90,200,0,5})` in 0.18 seconds, despite there being 15!=1,307,674,368,000 permutations.

[4]In many models, the variant actually corresponds exactly to the number of iterations.

```
out_sortedSequence <-- Sort_OP(p_set) =
  PRE p_set : POW(INTEGER) THEN
    VAR v_set, v_seq
    IN
      v_set := p_set; v_seq := [];
      WHILE v_set /= {}
      DO
        v_seq := v_seq <- min(v_set);
        v_set := v_set \ {min(v_set)}
      INVARIANT
        v_set : POW(p_set) & v_seq : iseq(p_set)
        & !i.(i : 1 .. size(v_seq)-1 => v_seq(i) < v_seq(i + 1))
      VARIANT card(v_set)
      END;
      ASSERT ran(v_seq) = p_set THEN out_sortedSequence := v_seq END
    END
  END
```

Figure 5.6: Implementation of a sorting algorithm.

algorithms on railway topologies we have modelled state machines instead of stateless query operations. However, the execution of these state machines is controlled by a single operation of an additional interface machine. A small disadvantage of using operations is that the output value of the operation can only be assigned to a variable and the operation can not be used as part of a set comprehension or quantification.

5.7.2 Recursive Functions

Recursive functions, which are supported by ProB [89], are a very effective way to compactly express certain kinds of algorithms. Figure 5.7 shows the selection sort algorithm modelled as a recursive function in B. By defining *Recursive_Sort* as an abstract constant we indicate that ProB should handle the function symbolically, i.e. ProB will not try to enumerate all elements of the function. The recursive function itself is composed of two single functions: one function defining the base case and one defining the recursive case. Note, that the intersection of the domains of these function is empty, and hence, the union is still a function.

```
ABSTRACT_CONSTANTS Recursive_Sort
PROPERTIES
  Recursive_Sort : POW(INTEGER) <-> POW(INTEGER*INTEGER)
  &
  Recursive_Sort =
                 λ(in).(in : POW(INTEGER) & in = {} | [])
                 \/
                 λ(in).(in : POW(INTEGER) & in /= {}
                            | min(in) -> Recursive_Sort(
                                           in \ {min(in)}))
```

Figure 5.7: Recursive sort function

However, there are certain constructs that are harder to write (and to read) using only the expression language of B, as it has no explicit support for let expressions and if-then-else by default. Nonetheless it is often easier to express a construct as a recursive function than it is to decompose the steps in order to express it as a machine. In general, the performance of a recursive function is slower compared to the operation and while loop approach.

Rather than using an explicit recursive call as in Figure 5.7, we can also use B's transitive closure operator to compute the fixpoint of a relation. For our example, let us define the relation `step = %(s,o).( s/={} | (s  {min(s)}, o <- min(s)))}` which encodes one recursive step of selection sort (s is the set to sort, o is the output sequence so far). For the set $in = \{4, 5, 2\}$ we can now compute `closure1(step)[{(in,[])}]` resulting in $\{(\{4, 5\} \mapsto [2]), (\{5\} \mapsto [2, 4]), (\varnothing \mapsto [2, 4, 5])\}$. As we can see, the result of sorting a set in can be obtained by calling `closure1(step)[{(in,[])}]({})`.

5.7.3 External Functions

B as a formal modelling language does not contain all concepts present in a programming language. For instance, in B there is no concept of a standard library that could provide mathematical functions such as sin, cos, etc. Other computations are difficult or impossible to express using only predicates and expressions, while others might be too slow to evaluate purely in B.

ProB offers a mechanism named `external functions` to add and expose new constructs to B. In our sorting example this might look as follows:

```
DEFINITIONS
 SORT(X) == [];
 EXTERNAL_FUNCTION_SORT == (POW(INTEGER) → seq(INTEGER))
```

The function `SORT` is implemented in Prolog within ProB's core and exposed in B as a definition. In order to define a syntactically correct `DEFINITION` we use the empty sequence as a dummy value ensuring type correctness. The second definition tells ProB the type of the external function. In general, external functions provide the best performance to execute specific computations. This is the case, because they remove the interpretation overhead since they are implemented in Prolog. For this reason they are opaque to the user and at this point in time it is not possible for users to define custom external functions.

5.7.4 Further Language Extensions

In the topology validation project we have introduced further language constructs that provide a uniform schema to write validation rules. Validation predicates are embedded in special `RULE` operations. Figure 5.8 shows a simplified schema of a `RULES_MACHINE` which contains several `RULE` operations and will be translated to an ordinary B machine. The result of a `RULE` operation can be stored by using the new `RULE_SUCCESS` or `RULE_FAIL(.)` keywords. The argument of the `RULE_FAIL(.)` keyword is the message reported in case of a rule violation. For each rule operation an ordinary variable is generated in the translated B machine containing the result of the rule evaluation (i.e. `"FAIL"`, `"NOT_CHECKED"` or `"SUCCESS"`). By using additional guards we are able to define dependencies between rules (using the new keyword `DEPENDS_ON_RULES`) or disable a rule if necessary. The model itself is non-deterministic in the sense that different rules can be executed at the same time if their guards are satisfied. Thus, we are not forced to define an explicit execution order of all rule operations and can use ProB's animation feature to conveniently execute a certain operation. To ease the writing of a rule we developed a new `FORALL` substitution which can be used to define an error message of a rule by conveniently accessing the variables of a universal quantification.

```
RULES_MACHINE Rules
SEES Features
OPERATIONS
 RULE rule1 = ...;
 RULE rule2 = ...;
 RULE rule3 =
  SELECT
   DEPENDS_ON_RULES(rule1, rule2)
   & Enabled(feature1) = TRUE
  THEN
   FORALL
    p1, p2
   WHERE
    P(p1,p2)
   EXPECT
    Q(p1,p2)
   THEN
    RULE_SUCCESS
   ELSE
    VAR errorMessage
    IN
     errorMessage := Exp(p1,p2);
     RULE_FAIL(errorMessage)
    END
   END
  END
END
```

```
MACHINE Rules
SEES Features
VARIABLES rule1, rule2, rule3
INVARIANT
 rule1 : {"NOT_CHECKED",
     "FAIL", "SUCCESS"}
 & rule2 : {"NOT_CHECKED",
     "FAIL", "SUCCESS"}
 & rule3 : {"NOT_CHECKED",
     "FAIL", "SUCCESS"}
INITIALISATION
 rule1 := "NOT_CHECKED"
 || rule2 := "NOT_CHECKED"
 || rule3 := "NOT_CHECKED"
OPERATIONS
 res,ce <-- rule1 = ...;
 res,ce <-- rule2 = ...;
 res,ce <-- rule3 =
  SELECT
   rule3 = "NOT_CHECKED"
   & rule1 = "SUCCESS"
   & rule2 = "SUCCESS"
   & Enabled(feature1) = TRUE
  THEN
   IF
    !(p1,p2).(P(p1,p2)
       ⇒ Q(p1,p2))
   THEN
    rule3 := "SUCCESS"
    || res := "SUCCESS"
    || ce := ""
   ELSE
    ANY p1,p2
    WHERE
     P(p1,p2) & not(Q(p1,p2))
    THEN
     VAR errorMessage
     IN
      errorMessage := Exp(p1,p2);
      rule3 := "FAIL"
      || res := "FAIL"
      || ce := errorMessage
     END
    END
   END
  END
END
```

Figure 5.8: Translation of a RULES_MACHINE to an ordinary B machine

5.8 Interaction With the Model

There are several ways the main software can interact with the validation model. Depending on the kind of application, one could animate or model check the B model, execute an operation or evaluate expressions or predicates (assertions) on a certain state of the model.

The ProB Java API (aka. ProB 2.0) provides facilities to use ProB in applications running on the JVM. Through this API it is possible to access the functionalities mentioned above and to translate B data types to and from appropriate Java types.

In case of the curriculum validation project, the tool itself is a Java application that embeds the model and ProB. We expose all features provided by the model as B operations that represent the public API of the model. These operations are evaluated, using ProB's animation facilities. Operations are executed with externally provided parameters to validate the different curricula. The validation operations return a list of variables that represent one possible choice of subjects to successfully finish a degree. Furthermore, we use the result computed for a feasible curricula to generate a PDF timetable for students with a recommend choice of subjects for their studies.

In the topology validation project the model is used as an independent validation tool with the goal to generate validation reports about the input data. Each engineering rule is modelled as one or more `RULE` operations containing the validation predicates (see Section 5.7.4). By using more than one `RULE` operation for an engineering rule the complexity of a natural language requirement can be decomposed into several simple and readable validation predicates. In contrast to listing all validation predicates as part of the `ASSERTIONS` section using `RULE` operations has the advantage that a B operation defines a clean interface to perform the evaluation of the individual rules, to access result values and counterexamples. In order to generate a complete validation report and to validate all possible rules, we construct a trace of the model using ProB's `execute` command until all operations are covered. By doing this, we eliminate the overhead which would be introduced by performing a complete model checking run on the non-deterministic model (i.e. evaluating an operation several times).

5.9 Configuration Management

Configuration management, i.e. how to reuse rules and infrastructure for similar or related projects which differ in very specific aspects, is very important in the context of data validation. For example, in the case of curricula validation, there are subtle differences amongst faculties in the overall structure or how the students choose classes. We have explored two different approaches to tackle this issue.

One approach is that of a Software Product Line (SPL) [32], where the system would create, from a selection of predicates and evaluation rules a machine that composes them according to a provided configuration. A further approach would be to search for and find a data representation and formulation of the validation rules that is general enough to be applied to more than one particular instance. Such a generic model can contain variation points to control specific aspects of the validation process that differ from project to project. For example, the rule in Fig. 5.8 is only tested when two particular features are selected.

In both projects we have settled for a combination of both approaches, automatically generating certain parts of our models and additionally configuring the generic parts.

5.10 Conclusion and Future Work

In this chapter we have presented two data validation projects where we have expressed the validation rules in B. Based on the experiences gathered and the similarities between the projects, we have discussed different relevant areas and presented our architecture and design decisions as well as possible alternatives.

We have identified the aspects of data validation that can be easily and elegantly expressed in B such as deriving intermediate data structures from the raw data, modelling complex algorithms while ensuring their correctness, and formalizing validation predicates which are close to natural language counterparts. Otherwise, we presented the points where we had to diverge from B by either using language extensions supported by ProB or by moving certain features outside of the B models, e.g. the data import. Moreover, we described a way to interact with the formal model and to build various applications on top of ProB.

In both projects ProB satisfies the respective requirements on performance and execution time. For the curriculum validation, ProB is able to detect conflicts among courses in an appropriate time, making interactive use on top of ProB possible. In the topology validation project there are no strict timing constraints. However, our B and ProB based approach is able to compete with a pre-existing validation tool written in an imperative language. In the next chapter we will perform a detailed comparison of this B based approach to other tools and languages.

The work on both projects has helped to push the development of ProB forward by highlighting performance bottlenecks that have since been resolved. Moreover, we added support for language constructs such as tree operators.

Due to the availability of higher-order data types, B can be used almost like a functional programming language. We have used this in particular to compute derived data. In the chapter we have also shown various limitations of B, and have presented some ways to overcome them. In the future, we would like to be able to use parts of B as a proper functional programming language. In that sense, we are considering adding polymorphic operators, as present in TLA$^+$, to provide a simpler way to structure predicates and allow the user to define new recursive operators. Moreover, we are pursuing an approach to embed parts of the mathematical B language into the Clojure programming language using native syntax and evaluating it with ProB, an approach comparable to aRby for Alloy [100].

6 Comparative Evaluation

6.1 Introduction

In the previous chapters we have discussed the use of B and ProB as tools not only to specify and validate software systems but also to formalize and solve complex constraint based data validation problems. The question remains: how does this high-level approach compare to other languages and tools that can be used in this domain. Is the formalization really simpler to understand and easier to modify than a comparable implementation to the same problem in a different formalism? Does a lower level representation affect the readability? Is the performance of the engine (either a solver, virtual machine or runtime) executing a formalization appropriate for the kind of problem and features of the language used? Appropriate here means that the cost of the added levels of abstraction is not too high with regard to evaluating a given representation.

Having a formalism that makes it easy to represent complex problems can be very useful. If the associated tool or runtime is not powerful enough to solve the represented problems in a reasonable amount of time for the abstraction level, the tool has little practical advantage. In that case the formalism is merely a vehicle of thought to structure a problem. On the other hand having a tool that can efficiently compute results for complex constraint based problems but only through a complex formalization process, that is tedious and very low-level, might also not always be an advantage, in particular when the problem at hand gets complex and its representation becomes unmanageable.

This duality can be found in the development of many modern programming languages, which are usually built around specific paradigms and try to provide the best compromise of performance and abstraction levels within their scope.

The goal of this chapter is to discuss, based on a simplified version of the case study discussed in Chapter 4, aspects of the implementation and performance of this problem using different formalisms and tools.

6.1.1 Scope

In order to assess the mentioned properties of different systems, we have selected some key features of the B model described in Chapter 4 that represent the interesting and challenging aspects of the problem and have implemented these in different formalisms. These features are the conflict detection and the unsatisfiable core (`unsat core`) computation.

The goal is to compare the different solutions with regard to two properties. These two are: first the *model's or implementation's complexity* and second *the runtime performance of the tool associated to each language or formalism.*

Conflict detection is the central feature of the B model created in the case study and is thus the key aspect to compare formalizations. The computation of an `unsat core` on the other hand was chosen because it is handled very differently by the selected tools. B does not have explicit support for it, but support has been added to ProB through an external function. Alloy provides built-in support for computing an `unsat core`. In languages like Prolog or Python it is not part of the language, but can be implemented using recursion. We want to compare the different approaches to implementing - where needed - and using the `unsat core` computation in the different tools.

6.2 Data Model for Evaluation

The comparison discussed in this chapter is based on the second version of our models, see Table 4.2 for a table highlighting the features in each version. This version presents a simpler data model, which is described below. It still contains all relevant validation aspects and steps to make it a meaningful problem for comparison.

6.2.1 Description of the Data Model

The model is based on the same entities as the third version described in Chapter 4. Below we will describe the data model used for this evaluation, which should clarify the differences to the more complex model described before. The different entities present in this data model, their relationships and the selection rules for the validation are visualized in Figure 6.1 as a feature model, see Section 4.2.4 for details on feature modelling.

In this model each **course** can be composed of a major and a minor or it can be a standalone course. E.g. the course `ger_ges` represents the combination of "**Ger**man Literature Studies" as major and "History" (**Ges**chichte) as minor.

Each **course** is composed of a set of **modules** where each **module** can be either mandatory or elective within a **course**, shown in Section (a) of Figure 6.1.

One important difference is that this model does not include the concept of **abstract units**, which was introduced for the third version of our models. The inclusion of abstract units adds a decision variable to our model and increases the branching. Without abstract units, **modules**, as can be seen in Section (b) of Figure 6.1, are composed of **units**, which themselves can be either mandatory or elective. **Units** on the other hand, as shown in Section (c) of Figure 6.1, are divided into **groups**. Each **group** is a set of **sessions** (Section (d) in Figure 6.1). Each of the **groups** within a **unit** is considered alternative to the others, in the sense that choosing any satisfies the **unit's** requirements. **Sessions** are the actual events, these are associated with the day and time when they take place. Additionally, each **session** has a rhythm that represents how often the **session** takes place. The rhythm can be weekly or biweekly (in even or odd numbered weeks). In this model the day and time information is represented using an atomic value that contains both pieces of information. This format was being used by one of the participating faculties and therefore used in the modelling. In Version 3 of the models we replaced this by separate day and time fields in the representation of **sessions**. The atoms representing day and time are composed of a letter, from a to f representing the time slot, and a number from 1 to 5 representing the day of the week. For example the atom $c2$ represents the third time slot on a Tuesday.

In this model one **module** can be associated with several **courses**. Also, a **unit** might be associated to more than one pair of **module** and **course**. The information in which semester a unit is available and whether it is mandatory or elective is associated to the triple of **course**, **module** and **unit**, e.g. A unit might be mandatory in one course and module but elective in another.

6.2.2 Validation

The goal of the validation process is to decide if there exists:

- a choice of **modules** (mandatory and elective),
- a choice of **units** in those **modules**,
- a choice of **semester** when to attend each **unit**,
- and a choice of one **group** for each **unit**.

Such that there are no scheduling conflicts among the **sessions** selected for the groups of units scheduled for the same semester.

The validation process works in a way similar to the process described for version 3 of the models. The process starts by selecting a **course** to be validated. In this version only one **course** is selected, since it already represents either a combination of major and minor or a stand-alone course.

To find a valid solution, the validation process must select all mandatory **modules** for the chosen **course** and a **course** course specific number of elective **modules**, which is shown in Section (a) of Figure 6.1.

Similarly, the validation must selected all mandatory **units** in each **module** and a **module** specific number of elective **units** must be chosen from each module as shown in Section (b) of Figure 6.1.

Each selected **unit** is assigned to a semester when it should be attended by the validation process. Possible choices of **semesters** depend on the **course** and **module** assignment. In each **module** of a **course** a **unit** might be available in different semesters.

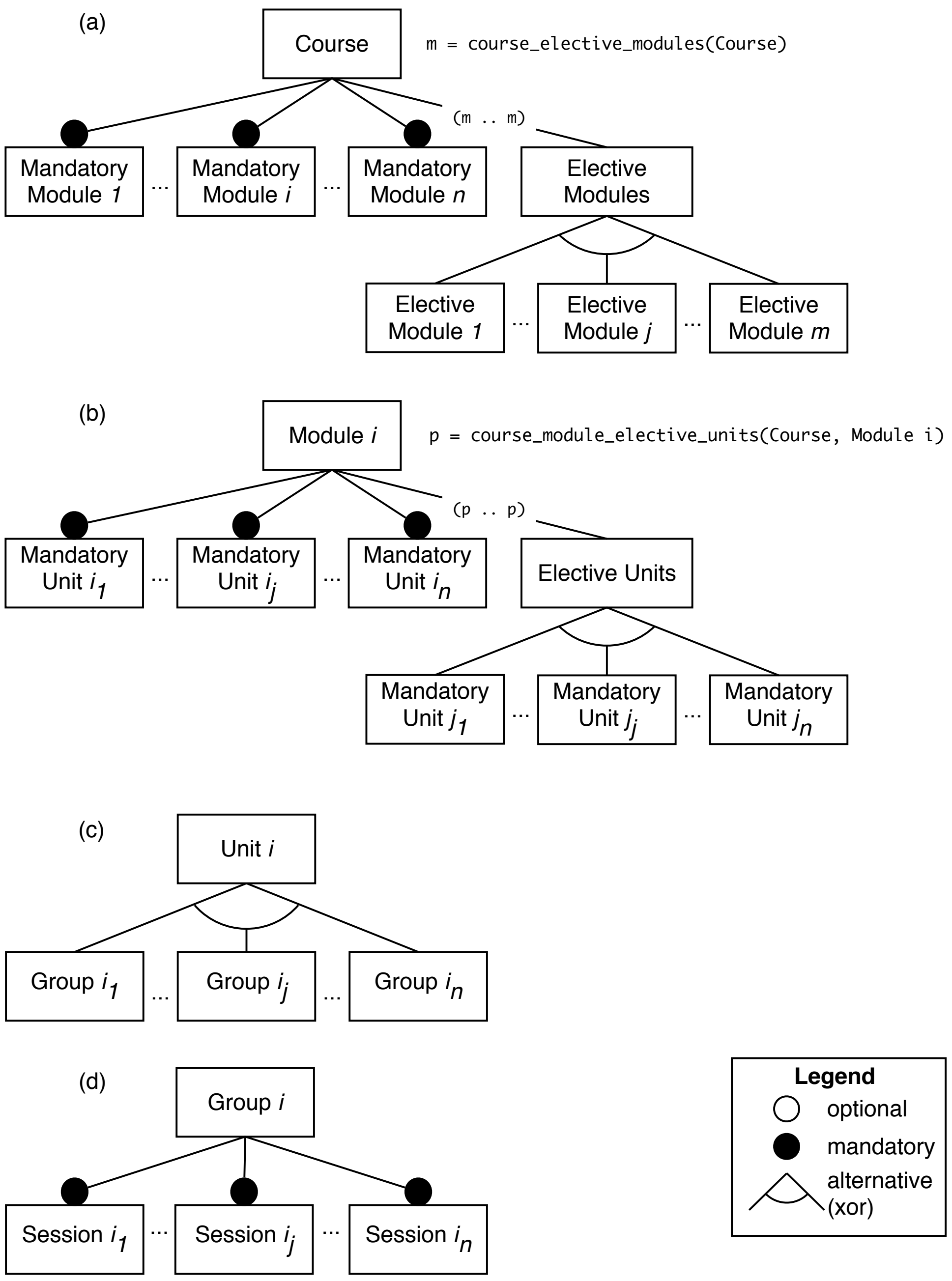


Figure 6.1: Feature diagram describing the validation and selection rules for the second version of our model.

Lastly, for each **unit**, as part of constructing a possible instance, a **group** is selected, as shown in Section (c) of Figure 6.1. For **units** assigned to the same semester, the validation must ensure, that there are no scheduling conflicts among the **sessions** of the **groups** selected in each **unit**.

The challenge for any implementation is not only to decide if a given assignment of values corresponding to the different described choices is correct, but also to find such a choice of **units** and **groups** for a **course** that is valid.

6.2.3 Data Sets

The evaluation will be performed on two data sets containing curriculum data provided by the faculty of *Arts and Humanities* (**AH**) and the faculty of *Business Administration and Economics* (**BAE**) at Heinrich Heine University Düsseldorf.

Each data set has different characteristics. The AH data set contains 67 different courses which not use the concept of modules, but contains units with many groups. The data set provided by the faculty of BAE contains 6 courses with many elective units and modules.

As an orientation for the reader we have computed a *complexity index* (**CI**) for every course in each data set that represents the number of possible instances - each instance stands for a unique selection of modules, units, groups and semesters - that each course has (feasible or infeasible). The worst-case, in a brute-force approach, would require each of these instances to be checked in order to determine if a course is feasible or not. Hence, the CI should give an idea about the complexity of each data set. Table 6.1 shows the different courses grouped by faculty, their feasibility, their CI, their number of mandatory and elective modules as well as the number of possible combinations of elective modules (Appendix D.1 contains this information for all courses in both data sets). Because the AH data set does not use the concept of modules, all course data is grouped in a single module. For the same reason the AH data set does not have elective modules and elective module combinations. Figure 6.2 shows, for each faculty, a plot of the courses ordered by their CI, highlighting how these vary and increase.

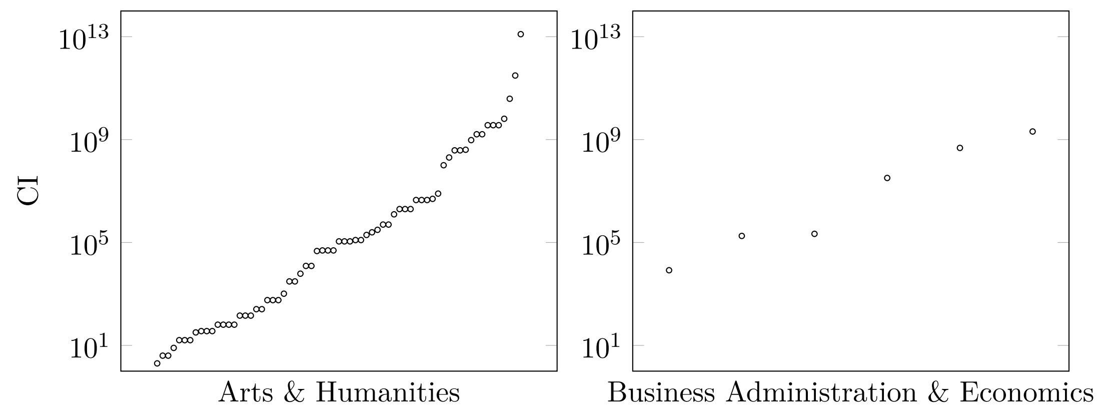


Figure 6.2: Plots of the courses in each faculty ordered by their *complexity index*. The *complexity index* is presented in a logarithmic scale.

A large CI does not necessarily mean finding a feasible instance will be slow, this rather depends on the used search strategy, but indicates the size of the search space. The CI can vary depending on the completeness of the provided data sets and the degree of freedom to choose units and groups for the students.

6.3 Languages and Tools

There are many interesting tools and languages worth considering for the evaluation of this case study. We have decided to restrict the selection to the following languages and tools in order to compare them to our B and ProB based approach:

- Prolog
- Prolog using clp(FD)
- SMT-LIB using Z3
- Alloy
- Python

We have selected these languages as they represent different language paradigms, programming concepts and computation models. Prolog [42] and Python [127] are general purpose programming languages, whereas Alloy [72] and SMT-LIB are modelling languages for

Table 6.1: *Complexity index*, feasibility, number of mandatory, elective and combinations of elective modules for the *Business Administration & Economics* data set and for the 25 courses with the largest *complexity index* values in the *Arts & Humanities* data set.

Course	Feasible?	CI	Mandatory Modules	Elective Modules	Elective Module Combinations
		Business Administration & Economics			
wichem_master	✓	8352	2	10	45
bwl_master	✓	180153	5	25	300
vwl_master	✓	217203	4	26	325
wichem_bachelor	✓	32265043	8	16	560
vwl_bachelor	✓	475079582	11	31	4495
bwl_bachelor	✓	2067530678	12	31	4495
		Arts & Humanities			
jap_jud	✗	497664	1	0	0
jud_ang	✓	1244160	1	0	0
ger_jap	✓	1990656	1	0	0
jap_ger	✓	1990656	1	0	0
jap_lin	✗	1990656	1	0	0
jap_kom	✗	4478976	1	0	0
jap_pol	✓	4478976	1	0	0
jap_soz	✓	4478976	1	0	0
ger_ang	✓	4976640	1	0	0
jap_jid	✗	7962624	1	0	0
ang_ges	✓	100776960	1	0	0
ang_inf	✓	201553920	1	0	0
jap_rom	✗	382205952	1	0	0
rom_jap	✗	382205952	1	0	0
ang_jud	✓	403107840	1	0	0
rom_ang	✓	955514880	1	0	0
ang_ger	✓	1612431360	1	0	0
ang_lin	✗	1612431360	1	0	0
ang_kom	✓	3627970560	1	0	0
ang_pol	✗	3627970560	1	0	0
ang_soz	✓	3627970560	1	0	0
ang_jid	✓	6449725440	1	0	0
jap_ang	✓	38698352640	1	0	0
ang_rom	✓	309586821120	1	0	0
ang_jap	✓	12538266255360	1	0	0

constraint solving and model finding tools. Prolog, Alloy and Z3 are declarative languages, while Python is an object-oriented and imperative language. Additionally, many Prolog implementations have built-in support for constraint logic programming (CLP) of which we will look at finite domain solving (clp(FD)). We explored using Constraint Handling Rules (CHR) [51] to model the problem, but abandoned this approach as it did not provide significant advantages over clp(FD) at the cost of a rather cumbersome encoding of the rules. Also, we explored the inclusion of TLA$^+$ [83] in the evaluation, but decided not to include it for two reasons: first, because the language is very similar to B in many aspects and second, because the available tools for TLA$^+$ are centred around model checking and proving characteristics of the model. Thus the tools do not offer any features that would allow to efficiently solve constraint based problems beyond brute-force [83, p. 232].

Each of the implementations is composed of two parts. These are a representation of the data structures trying to use the most adequate concepts in each language and a formalization of the timetable validation problem as described earlier.

6.4 Evaluation

In this section we will present implementations of the problem described above using the chosen languages. Each section follows the same structure of a brief introduction of the language and the tool, followed by a discussion about how we implemented our solver in each language, i.e. which language specific constructs we used. For each language we explain how the search is performed, in particular how we enumerate the variables to find models to check for feasibility. Additionally, we describe how the search for an unsatisfiable core is implemented in each tool. Lastly we present and analyse benchmark results for the feasibility check for each tool separately. At the end of this chapter we present a comparative discussion of the evaluated tools.

All benchmarks presented in this chapter were conducted by running, for each data set separately, the feasibility check for all courses. We ran each benchmark 10 times and report the average runtime for each course over the 10 runs. All benchmarks were run with a timeout of 30 minutes after which we cancelled the corresponding check. All benchmarks were run on an otherwise idle machine running Linux Mint 18, with 4 GB of RAM and a quad-core Intel Core i5 CPU running at 2.67 GHz.

6.4.1 Prolog

Prolog [42] is a declarative programming language, based on first-order logic originally created in 1972 [33, 78]. Since 1995 the central aspects of the language have been standardized by the International Organization for Standardization (ISO) [69]. Prolog programs are created by defining logical facts and inference rules. These are used to evaluate whether queries posted to the system are satisfiable and, if so, what variable bindings are needed for the query to be true.

It is interesting to consider Prolog in our evaluation for several reasons; first the constraint solver for B provided by ProB is implemented in SICStus Prolog. Prolog, as a programming language based on first order logic, makes it easy to express rule based systems as used in this case to verify the feasibility of a curriculum. Additionally, many Prolog systems have support for constraint programming which extends Prolog semantics very naturally. ProB makes use of these features in SICStus Prolog in order to implement parts of its constraint solving kernel [87].

Facts and rules are defined by means of Horn clauses, i.e. a disjunction where no more than one literal is not negated. A program is evaluated by posting a query, a logical expression that is evaluated for its satisfiability with regard to the defined rules and facts. For each part of a query, a Prolog system will check if it is satisfiable with regard to the defined facts and rules using a systematic search process, called resolution. In this process a element of the query will be matched against facts and rules by unification until a solution or a contradiction is found. Unification is one of the central concepts in Prolog, which is used to decide if two terms are structurally identical or, if they contain uninstantiated variables, can be made identical by binding their variables [121].

The default approach to enumeration in Prolog is the use of choice-points and backtracking. This has the advantage, compared to an imperative iteration approach, that it is not necessary to manually keep track of the state of the enumeration, since it is part of the language's execution model. In the recursive search performed by the Prolog engine, each time a logical variable is associated with a value a choice point is created. If a search fails the engine will backtrack to a choice point and bind variables to the next possible value until either a solution was found or all possibilities are exhausted. Coroutines extend this enumeration by executing the program normally, until a coroutine is called. When a coroutine is called and its condition, usually the instantiation of its variables, is satisfied execution continues normally. If the coroutine's condition is not satisfied its execution is blocked and only resumed when the condition is satisfied [28].

clp(FD) Constraint logic programming over finite domains is an extension to Prolog, supported by several Prolog implementations [27, 126], for reasoning about integers. clp(FD) can be used to declaratively encode combinatorial problems over discrete domains using integers.

clp(FD) usually provides operations to associate variables with a domain, or range of values as well as operations to put variables in relation to each other and further constraining their domains. The implementations themselves associate the variables and operations with efficient algorithms to propagate the information about the constraints with the goal to reduce the search space for possible solutions. The efficient propagation of constraints and the associated handling of a variable's domain are central features of constraint programming. These characteristics can make it a very efficient approach for certain families of problems. In many cases the constraints associated to variables can help drastically reduce the size of the universe in which a solution is searched for. In this evaluation we will consider the clp(FD) implementation as provided by SICStus Prolog [26].

Enumeration of Variables

When using clp(FD) variables are associated with a range of values, representing their domain. Ranges can be reduced by attaching constraints to the variables. Each constraint can reduce the number of values from the domain that are acceptable for the variable. Using clp(FD), predicates and operators are used to setup a problem instance, variables are not automatically bound to a specific value, with the exception when there is exactly one possible value. Variables are bound to values in a second step, called labeling. In the labeling process each variable is systematically assigned values from its domain. Labeling proceeds until either every variable is assigned a value and the problem instance is satisfied or all combinations of values are exhausted, in which case there is no solution to the problem. Labeling can follow different predefined strategies to enumerate variables, depending on the problem, for example first enumerating variables with the smallest domain or variables with the most constraints associated to them, etc.

```
search(X, X) :- X > 99999999.
search(X, Z) :- Y is X + 1, search(Y, Z).

search2(Y) :- Y in 1..9999999999, Y #> 99999999, labeling([ffc],
   [Y]).
```

Figure 6.3: Artificial example of a recursive and a clp(FD) based search.

Consider the following artificial example in Figure 6.3 comparing clp(FD) to a recursive search. In this example, the `search` predicate searches recursively for a number greater than 9999999999. It checks if the current argument is a solution and if not it tries the next number until it finds a solution and takes about 1500 milliseconds for `search(1, X)` on our benchmark machine. The `search2` predicate on the other hand uses clp(FD) operators to define the domain of the solution and to constrain valid solutions to be greater than 9999999999 taking only about 10 milliseconds on the same machine.

Unsatisfiable Core

Prolog does not provide built-in means to compute the unsatisfiable core of a predicate. Nevertheless Prolog's recursive execution model lends itself to implement such a search. A very simple (and unoptimized) implementation of a search for a minimal unsatisfiable core could implemented in Prolog as shown in Figure 6.4.

Implementation

We have created two different Prolog based implementations, one that relies on Prolog's standard execution model and one that uses clp(FD), coroutines and labeling to constrain the search space.

Data Representation Both implementations share the same data representation. Each type of entity is represented using facts, each fact contains a numeric id and all further attributes, for instance a course is represented using the (simplified) fact `course/2`, e.g. `course(56, 'jap_jid')` represents course number 56, with the name `jap_jid`.

```
:- use_module(library(lists)).

uc(_, [], Acc, Res) :- sort(Acc, Res), !.
uc(Predicate, Values, Acc, Result):-
  member(V, Values),
  delete(Values, V, Residue),

  append(Residue, Acc, Args),
  Call =.. [Predicate, Args],

  (call(Call)
    ->uc(Predicate, Residue, [V|Acc], Result)
    ; uc(Predicate, Residue, Acc, Result)
  ).


pred(List) :- sum(List, Result), Result < 10.

sum([], Acc, Acc) :- !.
sum([I|Rest], Acc, Result) :- Tmp is Acc + I, sum(Rest, Tmp, Result).

sum(List, Result) :- sum(List, 0, Result).


:- setof(R, uc(pred, [2,7,3,8,6,1], [], R), Res), print(Res), nl.
 [[1,2,7],[1,3,6],[2,3,6],[2,8],[3,7],[3,8],[6,7],[6,8],[7,8]]
```

Figure 6.4: Prolog implementation of a search for a minimal unsatisfiable core of a list of values with regard to a given predicate.

Sessions are represented using the `session/4` functor as follows `session(id, rhythm, duration, slot)`, where each possible time slot is represented using an atom for the day and time pair, e.g. `a1` if the first time slot on a Monday and `a2` is the first time slot on a Tuesday.

Relations are facts that reference the different entities by their identifiers. E.g. `cmu(56, 1, 64, 'm', 2)` represents the association of course 56 with module 1 and unit 64, which is mandatory (`'m'`) in the second semester.

Brute-Force First we will focus on the brute-force approach to finding a feasible timetable in Prolog. The search for feasible solution takes advantage of Prolog's built-in backtracking. We perform a depth first search for a solution, instantiating each of the required choices, e.g. modules, units, semesters, etc, to one of the corresponding possibilities. Once we have constructed a candidate instance we check it for conflicts to decide if it represents viable curriculum.

Every time a computed instance is not valid we backtrack to the closest point where we can construct an alternative solution (a choice point) and continue the computation from there. If the alternatives for a choice point are exhausted (we have tried all possible instantiations) we backtrack further to the next choice point. If all combinations of values have been evaluated and we haven't found a satisfying instance, we know the course is infeasible. Conversely, if we find a feasible instance we can stop the search and interpret the current values of the logical variables as the timetable instance. The search process starts from a given course, selecting one of the possible module combinations, then selecting the units, mandatory and elective, needed to complete the modules. For a choice of units, all semester combinations are enumerated and each is checked to see if there is a choice of groups that is feasible under that specific choice of semesters. The brute-force approach does not discard combinations that might fail for the same reason as a combination checked before and relies on trying all possible combinations to detect a valid combination. Additionally, it does not consider any specific order of the variables being enumerated. Choices of semesters and groups are represented using AVL trees, these provide a data structure for key value mappings [28] with logarithmic complexity on lookup and insert operations. This is the best available choice, since SICStus Prolog does not support maps or dictionaries. In each AVL tree a unit is mapped to a logical variable that is bound to a semester and a group respectively during the search. After a semester and a group have been chosen for each unit, we check those pairs of units assigned to the same semester for conflicts. For each pair of units we check if the sessions in the selected groups are compatible. To check if two groups are compatible we compute all pairs of sessions and check each for conflicts using the `no_conflict/2` predicate as shown in Figure 6.7. Sessions are provided as a list of pairs and checked one after the other.

clp(FD) The clp(FD) approach works similarly to the brute-force approach described above. They differ in how the enumeration and validation of instances is performed. In the clp(FD) based approach we split these steps into one that step sets up the problem and collects the domains of the enumeration variables and a second that does the enumeration. In our solver we combine clp(FD) and coroutines, using in particular the ability to constrain the domains of the variables representing unit and group choices.

In the first step - for a given course - we collect modules and units and create coroutines that will block for each pair of collected units. The coroutines will block on variables used to represent the semester and group choice, see Figure 6.5, and only resume their execution when these variables are bound. Besides creating the coroutines, we collect information about domains of the enumeration variables, i.e. the sets of possible semesters and groups for each unit.

Choices of unit, semester and groups are represented as AVL trees that map each selected unit to a CLP variable, whose domain is constrained to the available alternatives for that unit. Figure 6.6 shows this for the map that represents the group selection.

The CLP-variables in the AVL trees are used in the labeling step, which performs the enumeration. The selected labeling strategy enumerates variables with small domains and many constraints associated to them first. Using this strategy certain conflicts will be detected early in the enumeration discarding the selection of units altogether and backtracking to choose a new set of units. A conflict that will be detected early is for example two units only available in the same semester each containing exactly one group where both groups are in conflict. During the labeling process once a variable is bound the coroutines attached to it are activated and continue their computation. In this way we create a partial solution that is further evaluated as soon as concrete assignments

```
:- block check_semester_group_unit_pair(?, ?, ?, -, ?, ?, ?, ?),
         check_semester_group_unit_pair(?, ?, ?, ?, -, ?, ?, ?),
         check_semester_group_unit_pair(?, ?, ?, ?, ?, ?, -, ?),
         check_semester_group_unit_pair(?, ?, ?, ?, ?, ?, ?, -).
check_semester_group_unit_pair(Unit1, Semester1, Group1,
                               Unit2, Semester2, Group2) :-
  Semester1 == Semester2,
  unit_group_sessions(Unit1, Group1, Sessions1),
  unit_group_sessions(Unit2, Group2, Sessions2),
  pair(Sessions1, Sessions2, Sessions),
  no_conflict(Sessions).

check_semester_group_unit_pair(_Unit1, Semester1, _Group1,
                               _Unit2, Semester2, _Group2) :-
  Semester1 \= Semester2.
```

Figure 6.5: Predicate to check unit compatibility with `block` annotations to create coroutines. Coroutines will wait on variables marked with `-` in the annotations.

```
group_choice(Units, GroupChoice) :-
  (foreach(U,Units), foreach(UGC,_GroupChoice) do
    (
      data:unit(U, _, _, UnitGroups),
      !,
      findall(I, member(group(I, _, _), UnitGroups), Groups),
      list_to_fdset(Groups, FDGroups),
      GC in_set(FDGroups),
      UGC = U-GC
    )
  ),
  list_to_avl(_GroupChoice, GroupChoice).
```

Figure 6.6: Predicate that collects possible groups for all selected units and created clp(FD) variables with constrained domains in an AVL tree.

```
no_conflict([]).

no_conflict([Session1-Session2|Sessions]) :-
  data:session(Session1, _, R1, Slot1),
  data:session(Session2, _, R2, Slot2),
  (Slot1 \= Slot2
    -> true
    ; R1= rhythm(Rhythm1), R2=rhythm(Rhythm2),
       Rhythm1 > 0,
       Rhythm2 > 0,
       Rhythm1 \= Rhythm2
    ),
  no_conflict(Sessions).
```

Figure 6.7: Checking for a potential conflict between two sessions.

become known. If the validation of a set of variables fails, Prolog will backtrack to a previous choice point, restoring the coroutines. The labeling process will try a new instance until a solution is found or all combinations have been checked. Once semester and group have been chosen for a unit the blocked coroutine can be triggered and can check the now known values of semester and group to detect if the choices for both units are compatible using the `no_conflict/2` predicate discussed before.

Results

We have based our benchmarks on SICStus Prolog. SICStus Prolog [26][1] is a commercial Prolog System developed at the Swedish Institute of Computer Science and compatible with the Prolog ISO standard [69]. Currently available in version 4.3.5, which is also the version used for our evaluation. There are many alternative Prolog implementations available, for example SWI Prolog [128], XSB Prolog [122], YAP Prolog[2] among many others. We have chosen SICStus Prolog for two reasons: first, it is one of the fastest Prolog engines and second, ProB is built on top of it, giving us the possibility to compare the both of them.

[1]`https://sicstus.sics.se` - [Online; accessed 31-March-2017]
[2]`https://www.dcc.fc.up.pt/~vsc/Yap/` - [Online; accessed 31-March-2017]

Table 6.2 shows results for the brute-force and clp(FD) based solvers. The table shows results for the BAE data set and for the 25 courses with the largest CI in the AH data set. All results for both versions are presented in Appendix D.2. Figure 6.9 shows the results of the clp(FD) and the brute-force solver for the infeasible courses in the AH data set, while Figure 6.8 shows the results for the feasible courses in this data set. Figure 6.10 shows the results for the clp(FD) and the brute-force solver on the BAE data set. All diagrams show the runtime of the solver for each course on a logarithmic scale. Courses are ordered increasingly by their CI.

For the brute-force benchmarks we have excluded results for courses that had not completed the feasibility check after 30 minutes, and marked them with a `timeout` in Table 6.2. These courses, as can be seen, correspond to instances with large CI values that are both feasible and infeasible but have a very large search space.

Table 6.2: SICStus brute-force and clp(FD) results for the *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set.

Course	Feasible	Brute-Force		clp(FD)		CI
		Runtime[‡]	SD	Runtime[‡]	SD	
Business Administration & Economics						
wichem_master	✓	1	±3.16	2	±4.22	8352
bwl_master	✓	1242	±55.94	2	±4.22	180153
vwl_master	✓	128	±4.22	0	±0.00	217203
wichem_bachelor	✓	timeout[†]	-	41	±3.16	32265043
vwl_bachelor	✓	timeout[†]	-	4	±5.16	475079582
bwl_bachelor	✓	timeout[†]	-	1237	±68.00	2067530678
Arts & Humanities						
jap_jud	✗	40200	±324.48	709	±18.53	497664
jud_ang	✓	4362	±37.06	0	±0.00	1244160
ger_jap	✓	22	±4.22	495	±21.21	1990656
jap_ger	✓	21	±3.16	496	±19.55	1990656
jap_lin	✗	212611	±1553	147	±6.75	1990656
jap_kom	✗	890961	±6464.23	0	±0.00	4478976
jap_pol	✓	135	±7.07	195	±11.79	4478976
jap_soz	✓	66151	±516.08	2	±4.22	4478976
ger_ang	✓	4	±5.16	2	±4.22	4976640
jap_jid	✗	810075	±3793.82	0	±0.00	7962624

[‡] Runtime in milliseconds.

[†] Each timeout represents a runtime of 30 minutes.

Table 6.2: SICStus brute-force and clp(FD) results for *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set.

Course	Feasible	Brute-Force		clp(FD)		CI
		Runtime[‡]	SD	Runtime[‡]	SD	
ang_ges	✓	`509267`	±`5046.97`	`3`	±`4.83`	100776960
ang_inf	✓	`timeout`[†]	`-`	`3`	±`4.83`	201553920
jap_rom	✗	`timeout`[†]	`-`	`0`	±`0.00`	382205952
rom_jap	✗	`timeout`[†]	`-`	`3`	±`4.83`	382205952
ang_jud	✓	`timeout`[†]	`-`	`0`	±`0.00`	403107840
rom_ang	✓	`13766`	±`78.34`	`6`	±`5.16`	955514880
ang_ger	✓	`33113`	±`331.70`	`0`	±`0.00`	1612431360
ang_lin	✗	`timeout`[†]	`-`	`1`	±`3.16`	1612431360
ang_kom	✓	`timeout`[†]	`-`	`0`	±`0.00`	3627970560
ang_pol	✗	`timeout`[†]	`-`	`2`	±`4.22`	3627970560
ang_soz	✓	`timeout`[†]	`-`	`4`	±`5.16`	3627970560
ang_jid	✓	`timeout`[†]	`-`	`4`	±`5.16`	6449725440
jap_ang	✓	`timeout`[†]	`-`	`1225`	±`30.64`	38698352640
ang_rom	✓	`timeout`[†]	`-`	`10`	±`4.71`	309586821120
ang_jap	✓	`timeout`[†]	`-`	`104041`	±`505.84`	12538266255360

[‡] Runtime in milliseconds.

[†] Each timeout represents a runtime of 30 minutes.

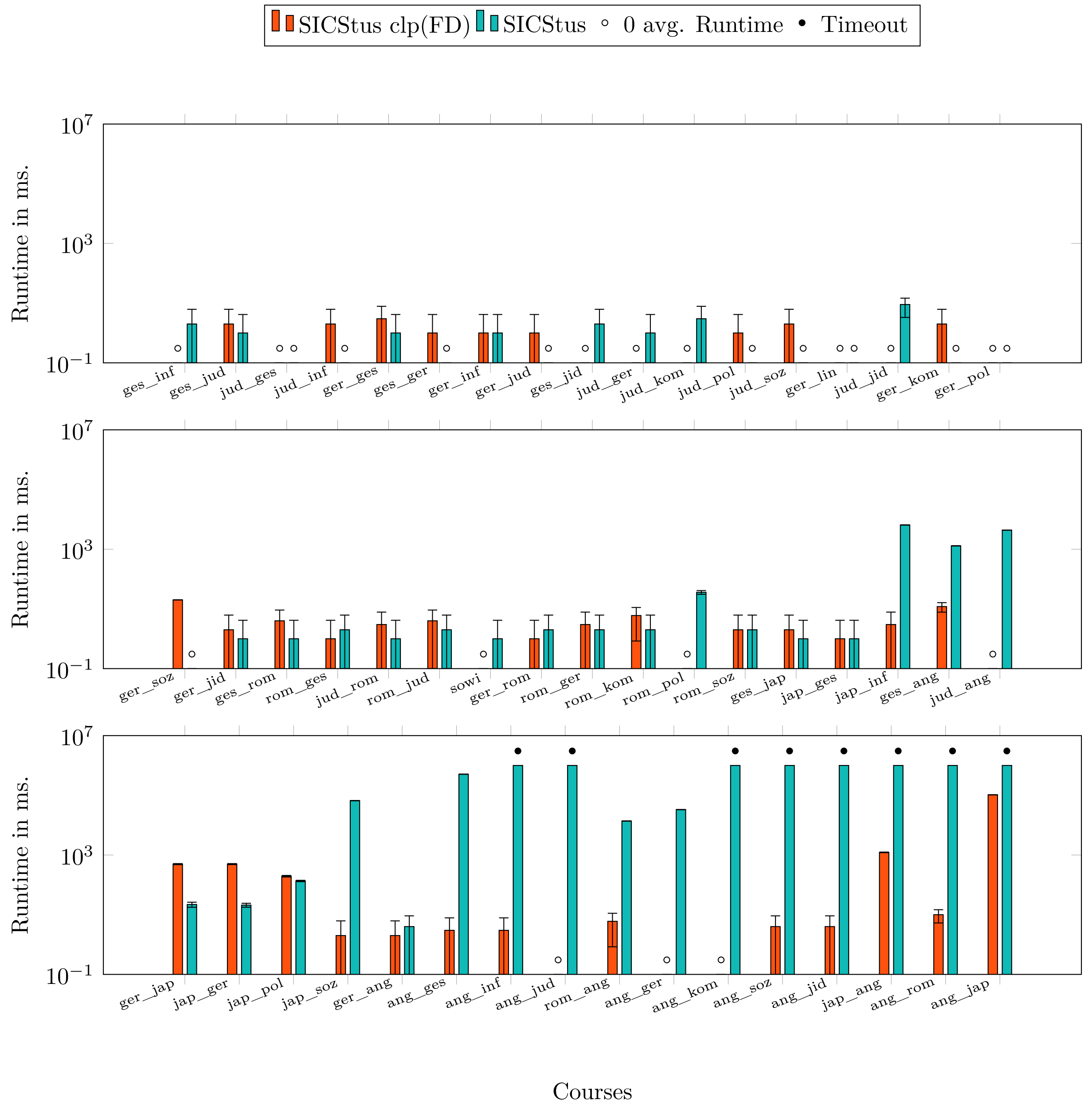


Figure 6.8: Prolog clp(FD) and brute-force results for **feasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*.

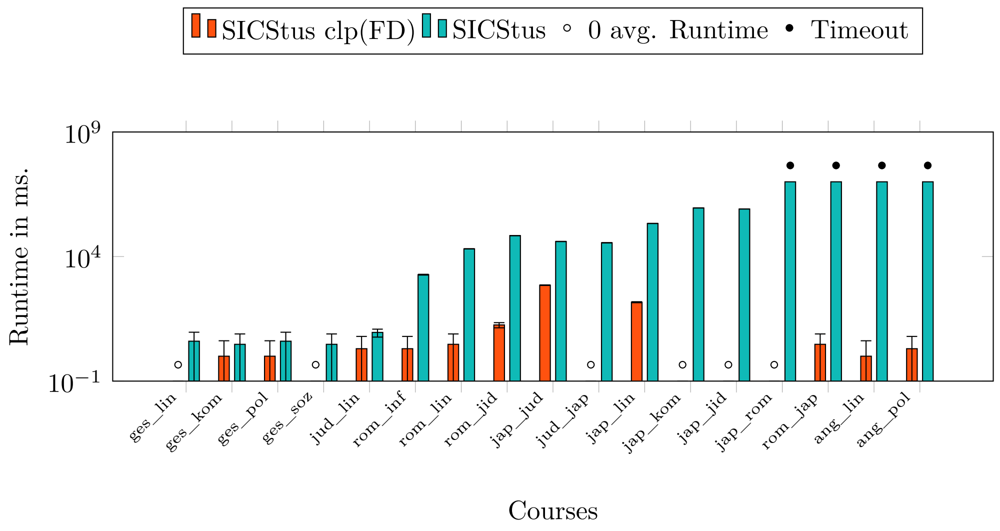


Figure 6.9: Prolog clp(FD) and brute-force results for **infeasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*.

Discussion Table 6.3 presents a summary of the results for both Prolog based implementations. The table shows that the brute-force approach can solve 55 of the instances in the AH data set and 3 of the 6 in the BAE data set. On the other hand the clp(FD) based approach finds a solution for all instances in both data sets. For the brute force solver, without considering the checks that were cancelled after 30 minutes, the median runtime is 0 milliseconds on the AH data set, taking them into account the median, as reported in Table 6.3, is 10 milliseconds over all benchmarks runs. On the BAE data set the median is 130 milliseconds, although we had to exclude 3 of the 6 courses, since they did not yield a result within 30 minutes. The median raises to about 15 minutes when considering the timeouts.

Our brute-force approach works rather well for instances with small CI values. Even for small instances it can be seen that deciding whether an instance is infeasible generally takes longer than deciding if is feasible. This is the case because the validation has to explore all possible combinations in order to decide that an instance is infeasible. For instances with larger CI values the performance degrades quickly. For example, the slowest result, that did not timeout, was `jap_kom` with a CI of 4478976. The validation took, on average over the 10 benchmark runs, about 15 minutes to detect that this instance is not feasible.

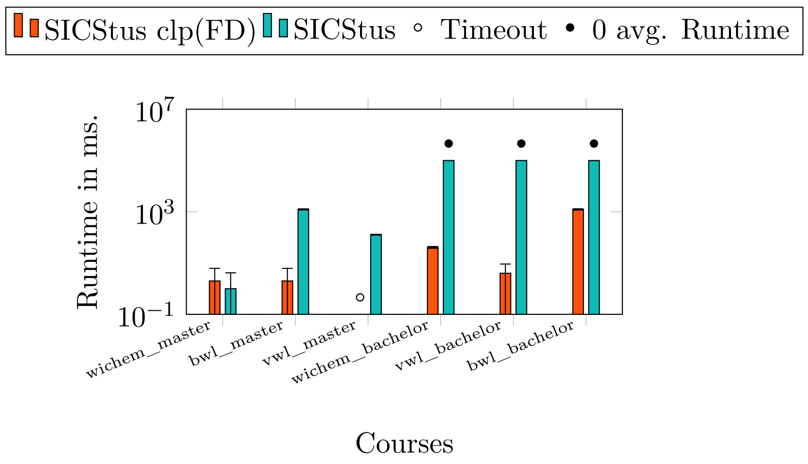


Figure 6.10: Prolog clp(FD) and brute-force results for courses in the *Business Administration & Economics* data set ordered by increasing *complexity index*.

Table 6.3: Summary of results for both Prolog based implementations.[†]

		Arts & Humanities		*Business Administration & Economics*	
		Brute-Force	clp(FD)	Brute-Force	clp(FD)
Results	# Feasible	42	50	3	6
	# Infeasible	13	17	0	0
	# Timeouts	12	0	3	0
RT[‡]	Average	362922.19	1603.79	900228.50	214.33
	Median	10.00	0.00	900700.00	0.00

‡ Runtime in milliseconds.
† We have taken timeouts into account when computing the average and median for each tool. Each timeout represents a runtime of 30 minutes.

In contrast, even the rather restricted use of clp(FD), of collecting the domain information and labeling with a specific strategy has a major impact on the runtime of this problem. Checking variables with small domains first and checking pairs of units as soon as their groups and semesters are known has a big advantage compared to completely instantiating a candidate solution before checking it and having no particular control over the order of the enumeration. The clp(FD) based approach finds a solution to all instances, with a maximum runtime of about 1 minute and 45 seconds for the `ang_jap` course, which is also the course with the largest CI value.

On instances with large CI values the performance benefit of the clp(FD) based approach is particularly notorious. Of the 25 results with the largest CI values in the AH data set the clp(FD) approach is faster on 23 of them, with the two exceptions being `ger_jap` and `jap_ger`. The brute-force approach did not yield a result before the timeout limit in 12 of these 25 cases. The constraint based solution can check 56 of the 76 AH courses and 4 of the 6 BAE courses in less than 10 milliseconds on average, compared to the 36 and 1 courses for the brute-force solver respectively. The median runtime for the clp(FD) based solver on both data sets is 0 milliseconds (considering SICStus' time resolution of 10 milliseconds).

With the regard to the implementation, there is no big difference between both approaches, certain parts are even shared between both. When using clp(FD) the implementation is slightly more complicated. Consider for instance the previously discussed Figure 6.6, where we specifically create an AVL tree of group-choice variables and associated them to their corresponding domains. Separating the collection, the search and the management of the variables' domains, as well as declaring the coroutines that wait on the enumeration of the variables leads to a more complex implementation, but with greater control over the search process.

The results collected for both Prolog based implementations show that, with regard to the runtime, it pays off to have tight control over the decision which parts of the search tree are explored and how variables are enumerated instead of just trying all possible combinations.

A possibility for improving the clp(FD) based approach would be to avoid waiting on all enumeration variables for a pair of units at the same time. One alternative would be to create constraints on the semester choice for all pairs of units; if the choice is equal we would constrain (using reification) the group choice such that it is conflict free, this approach could also be implemented using CHR [51].

6.4.2 SMT-LIB/Z3

Next we will discuss Z3 [39], which is a theorem prover based on SMT (satisfiability modulo theory) developed by Microsoft Research. Z3 was built with software verification in mind and has been used for this purpose in many projects inside and outside of Microsoft. Some examples are Pex [123], the Boogie verifier [13] among many others. Z3 has also been applied to other domains, such as the verification of network configurations [21].

Z3 supports several input formats, of which we decided to use the SMT-LIB [14] language to encode our timetable validation problem. SMT-LIB aims to be a common input language for SMT solvers and thus portable across tools. SMT-LIB is a LISP-like language with a prefix notation that is used to describe models by declaring sorts and data types, and creating assertions about the defined entities.

SMT is an extension of the SAT (boolean satisfiability) problem. The satisfiability problem is the problem of deciding if there is a truth assignment for the variables in a given boolean formula such that this formula (usually in conjunctive normal form) is true. The SAT problem itself is NP-complete [34] but there are efficient solvers that can handle a wide range of instances of this problem such as MiniSat [46], Lingeling, Plingeling and Treengeling [18], among others. Thus, suitable for use in different areas of computer science such as hardware verification [58], model checking [19], planning problems [76], etc. SMT extends SAT by adding to each boolean variable in the formula an interpretation in a theory. Theories might be integer arithmetic, bitvectors, arrays, etc. A truth assignment in the SAT formula then implies the satisfiability of an expression in one of the used theories. Figure 6.11 shows the same artificial example from Figure 6.3 presenting an SMT-LIB model for finding a value for the constant a that is greater than 99999999.

```
(declare-const a Int)
(assert (> a 99999999))
(check-sat)
(get-model)

Output:
sat
(model
  (define-fun a () Int
    100000000))
```

Figure 6.11: Example of a simple SMT-LIB model with one constant and one assertion.

```
(set-option :produce-unsat-cores true)
(declare-const a Int)
(assert (! (> a 0) :named positive))
(assert (! (> a 99999999) :named gt))
(assert (! (< a 10) :named lt))
(check-sat)
(get-unsat-core)

Output:
unsat
(gt lt)
```

Figure 6.12: Example of an unsatisfiable SMT-LIB model and of the unsat-core computation.

Unsatisfiable Core

Z3 provides native support for computing the unsat core of an unsatisfiable formula. To enable support for this, the `:produce-unsat-cores` option has to be set using `(set-option :produce-unsat-cores true)`. One important aspect of the unsat-core computation in Z3, is that Z3 only tracks named assertions when computing the unsat core for a problem and minimizes these. Figure 6.12 shows a simple example of an unsatisfiable model based on Figure 6.11. Two of the three assertions lead to a conflict, since the assertions $a < 10$ and $a > 99999999$ cannot be true at the same time.

```
(declare-datatypes () (
  (Slot a1 a2 a3 a4 a5 b1 b2 b3 b4 b5 c1 c2 ... g4 g5)
  (Semester sem1 sem2 sem3 sem4 sem5 sem6)))
```

Figure 6.13: SMT-LIB declarations of data types used to represent time slots and semesters.

```
(declare-datatypes (SID Slot Rhythm) ((SESSION (session (id SID)
   (slot Slot) (rhythm Rhythm)))))
(define-sort Session () (SESSION SID Slot Rhythm))
(define-fun s1 () (Session) (session sid1 a2 r0))
...
```

Figure 6.14: SMT-LIB data type and sort declarations used to represent **sessions**.

Implementation

To evaluate Z3 we have created a generic template, that combined with our benchmark data generates an SMT-LIB model for the data set. Our solver implementation does not use a specific logic, but relies on the Z3 default mode, using uninterpreted functions, algebraic data types as features of the different theories provided by Z3.

We represent the different sets of values used in the model as SMT-LIB data types. Figure 6.13 shows examples of the definition of the data types used to represent the different slots and semesters.

Sessions are represented as constants of a sort `Session` that represents a structure containing all relevant attributes of a session, as shown in Figure 6.14.

In our model we represent the different choices needed to characterize a valid timetable as boolean functions. These functions map the values in the corresponding domain, e.g. units, to true if the value is part of a feasible timetable and to false otherwise.

In some cases the problem requires the choice of a set of values with a certain cardinality from a given set, e.g. to choose n out of m elective modules. Z3 does not have native support for sets. As an alternative we represent this choice as a boolean function, see Figure 6.15 for an example. Instead of using the built-in boolean sort we use the integers 0 and 1. To express the cardinality constraints for a given instance of the function

```
(declare-fun unit-choice ((UID)) Int)
(assert (forall ((u UID)) (or (= (unit-choice u) 0)
                              (= (unit-choice u) 1))))

; choice of a module implies the choice of mandatory units
(assert (! (implies (= (module-choice mod1) 1)
            (= (unit-choice uid35) 1)) :named mod1_uid35-choice))

; expressing the choice of two elective units out of three for
   module mod1
(assert (implies (= (module-choice mod1) 1)
  (= 2 (+ (unit-choice uid26)
          (unit-choice uid28)
          (unit-choice uid27)))))
```

Figure 6.15: SMT-LIB function and assertions used to represent and constrain the **unit** choice.

the sum of the values in the range of the function must be equal to the requested set cardinality. In other words, if we only want to select three elements of a given domain, we assert that the sum of all function values for that domain is equal to three, hence only three elements can be mapped to 1 in any valid model.

The validation predicate is defined as an assertion, shown in Figure 6.16, that compares pairs of sessions in the data. Any two distinct sessions, that have been chosen, denoted by `(= (session-choice x) true)` in the current instance and assigned to the same semester have to satisfy the predicate defined in the function `compatible`, i.e. be free of binary conflicts.

One disadvantage of the SMT-LIB encoding is, that it does not allow to clearly separate data from validation rules. The lack of high-level and higher-order data structures makes the encoding complex, although it is for the most part rather straight forward. Certain constructs have to be made explicit, in order to achieve what could otherwise be expressed by traversing nested data structures and deriving constraints (see Figure 6.15).

This basically requires unrolling data structures in the generated model, that could otherwise be traversed as part of the validation. This leads to a model where more things

```
(define-fun same-rhythm ((a Rhythm) (b Rhythm)) (Bool)
  (or (= a b) (= a r0) (= b r0)))

(define-fun compatible ((a (Session)) (b (Session))) (Bool)
  (=> (same-rhythm (rhythm a) (rhythm b))
        (not (= (slot a) (slot b)))))

(assert (! (forall ((x (Session)) (y (Session)))
  (=>
    (and
      (distinct x y)
      (session-choice x)
      (session-choice y)
      (= (semester-choice (session-unit x)) (semester-choice
          (session-unit y))))
    (compatible x y))) :named session-conflict))
```

Figure 6.16: SMT-LIB assertion and predicates used to check pairs of sessions for conflicts.

are made explicit, hopefully saving computations for the solver, but at the same time making the generated model larger and more difficult to follow, debug and change.

Unrolling certain relations leads to an encoding that has explicit dependency rules for the data. E.g. stating that if a specific unit is chosen, only the n specific groups linked to it are allowed to be chosen for that unit, and all others are not (see Figure 6.17).

Enumeration Our validation predicate contains quantifiers, and although Z3 is not a decision procedure for first order logic, Z3 is able to validate many formulas containing quantifiers.[3] Hence we rely on the Model Based Quantifier Instantiation mode (`mbqi`) [54] to find models for our formula. This approach, described as a "counter-example based refinement loop"[3] works by creating models and checking them against the provided formulas until a satisfying model has been found or the search space is exhausted.

Using the `mbqi` evaluation mode the session-conflict predicate is instantiated with pairs of previously defined session values that are then checked for compatibility.

[3] `http://rise4fun.com/z3/tutorialcontent/guide` - [Online; accessed 24-January-2017]

```
(assert (ite (= (unit-choice uid1) 1)
  (xor (group-choice gid1) (group-choice gid2)
       (group-choice gid3) (group-choice gid4)
       (group-choice gid5) (group-choice gid4))
  (not (or
   (group-choice gid1) (group-choice gid2)
   (group-choice gid3) (group-choice gid4)
   (group-choice gid5) (group-choice gid6)))))
```

Figure 6.17: SMT-LIB of an assertion constraining the group choice for a unit to one of its groups in case it is selected.

Table 6.4: Summary of results for Z3.†

		Arts & Humanities	*Business Administration & Economics*
Results	# Feasible	50	6
	# Infeasible	17	0
	# Timeouts	0	0
RT‡	Average	17.126	97.904
	Median	20.035	95.995

‡ Runtime in seconds.
† We have taken timeouts into account when computing the average and median for each tool. Each timeout represents a runtime of 30 minutes.

SMT-LIB provides very limited support to control the enumeration, by attaching patterns to quantifiers ensuring, that the quantifier is instantiated when there are terms in the search context that match the pattern.[3] In our experiments these annotations did not improve the runtime of our benchmarks.

Results

For our evaluation we have used version 4.5.0 of Z3. Table 6.5 presents selected results collected from running Z3 on both data sets, all collected results can be found in Appendix D.3. A graph presenting the results for the faculty of BAE can be found in Figure 6.18. The results for the AH data set are presented in two diagrams, Figure 6.19 shows the results for the feasible courses in the data set and Figure 6.20 for the infeasible ones.

As can be seen, Z3 is significantly faster in detecting infeasible instances with a runtime of about one second in all these cases, independent of the CI. On the other hand, it does not perform very well on feasible instances. The longest runtime for a feasible instances on the AH data set is on average about 35 seconds for the `jap_ang` course which is associated to the third largest CI value of 38698352640. For the feasible courses in the BAE data set the averages range from 83 to 114 seconds with an average over all benchmarks of 98 seconds. From the feasible courses in the AH data, none takes less than 12 seconds to validate, with `ges_ger` being the course with the shortest average runtime of 12.15 seconds and a CI value of 16. None of the feasible courses among those with 25 largest CI values (see Table 6.5) takes less than 19 seconds to validate.

For infeasible courses the validation is significantly faster, e.g. the validation of the course `jap_jud` takes 1.14 seconds, which is also the slowest validation time for any infeasible course in the AH data set.

In Table 6.4 we have summarized the results for this implementation. The table shows that Z3 is able to find a solution for all instances without any timeouts and summarizes the average and median runtimes of the Z3 implementation on both data sets.

In general the median of 20 seconds with a top runtime of 35 seconds on the AH data set as well as a median of 96 seconds and a top runtime of almost 114 seconds on the BAE data set shows that Z3 does not perform very well on this search problem. On the other hand, it is to be noted that the behaviour shown by Z3 is very consistent across the different instances for feasible and in particular infeasible courses.

One possible reason for the performance could be that we tried to stay as high-level as possible in our formalization using SMT-LIB, in order to keep the model readable. A lower level encoding, maybe using the bitvector theory as done by Demirović and Musliu [41] for high-school timetables with the goal of optimizing them, might be an approach to achieve better results for the simpler validation task. This approach would require a more advanced translation process from the problem instance to an SMT-LIB model. Another issue with SMT-LIB, also discussed by Demirović and Musliu [41], is finding an efficient method to express cardinality based constraints which they solved using bitvectors and defining a method to ensure at-least and at-most n bits are set. As described before, we solved this using the sum of the values in the range of a function that maps to 0 or 1.

Table 6.5: Z3 results for the *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set.

Course	Feasible	Runtime in s.	SD	CI
		Business Administration & Economics		
wichem_master	✓	82.735	±0.616	8352
bwl_master	✓	86.898	±0.655	180153
vwl_master	✓	87.151	±3.321	217203
wichem_bachelor	✓	106.389	±4.293	32265043
vwl_bachelor	✓	113.974	±1.015	475079582
bwl_bachelor	✓	110.275	±0.468	2067530678
		Arts & Humanities		
jap_jud	✗	1.141	±0.164	497664
jud_ang	✓	22.953	±0.290	1244160
ger_jap	✓	25.471	±0.270	1990656
jap_ger	✓	26.449	±0.837	1990656
jap_lin	✗	0.972	±0.063	1990656
jap_kom	✗	0.984	±0.101	4478976
jap_pol	✓	29.284	±0.111	4478976
jap_soz	✓	27.646	±0.107	4478976
ger_ang	✓	19.741	±0.356	4976640
jap_jid	✗	0.952	±0.004	7962624
ang_ges	✓	22.275	±0.722	100776960
ang_inf	✓	28.585	±0.726	201553920
jap_rom	✗	0.982	±0.115	382205952
rom_jap	✗	1.012	±0.106	382205952
ang_jud	✓	25.903	±0.340	403107840
rom_ang	✓	30.541	±0.121	955514880
ang_ger	✓	21.313	±0.861	1612431360
ang_lin	✗	0.947	±0.007	1612431360
ang_kom	✓	26.025	±0.279	3627970560
ang_pol	✗	0.974	±0.071	3627970560
ang_soz	✓	27.888	±0.124	3627970560
ang_jid	✓	27.597	±0.215	6449725440
jap_ang	✓	35.044	±1.176	38698352640
ang_rom	✓	29.614	±0.074	309586821120
ang_jap	✓	32.141	±0.765	12538266255360

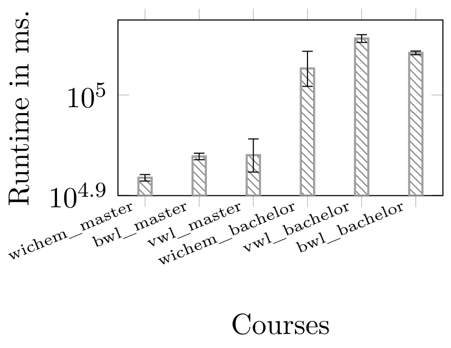


Figure 6.18: Z3 results for courses in the faculty of *Business Administration & Economics* data set ordered by increasing *complexity index*.

An potential approach to improve the performance of our solver would be to replace the functions mapping an entity to a boolean with a variant that uses a bitvector to track the selection state. However, initial experiments using bitvectors instead of a function mapping to 0 and 1 did not show an improvement for infeasible courses and even showed worse performance on feasible courses. In our experiment we used a bitvector, where each bit represents a specific entity, to express the selection state and the total number of set bits to express the cardinality constraints. E.g. for groups selection: if the 5th bit in the group-choice bitvector is set, our interpretation is that the group with the ID 5 is part of the current instance.

Another possible explanation is the use of universal quantification and of the `mbqi` evaluation mode that can defeat certain optimizations provided by Z3.

With regard to the implementation, the Z3 encoding is rather straight forward, but the lack of higher order data structures makes the encoding cumbersome, having to unroll nested data sets and defining all relationships explicitly. In general encoding problems in SMT-LIB leads to decomposing the problem into the low-level concepts available in Z3. In this regard Z3 might be a viable target for a translation or an automatic mapping from a higher level formalism, but makes the manual encoding of this kind of problems difficult.

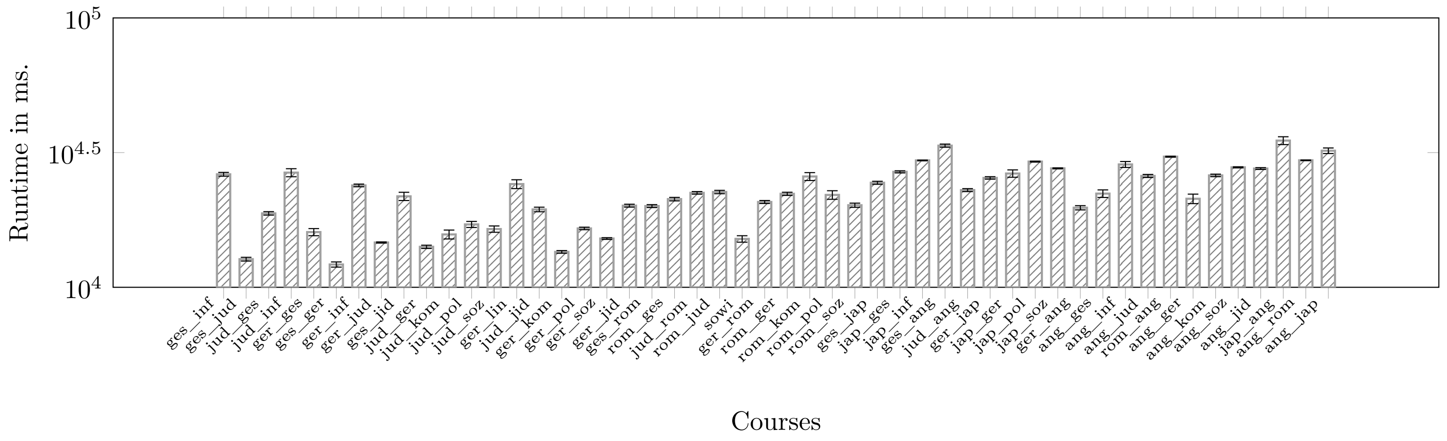


Figure 6.19: Z3 results for **feasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*.

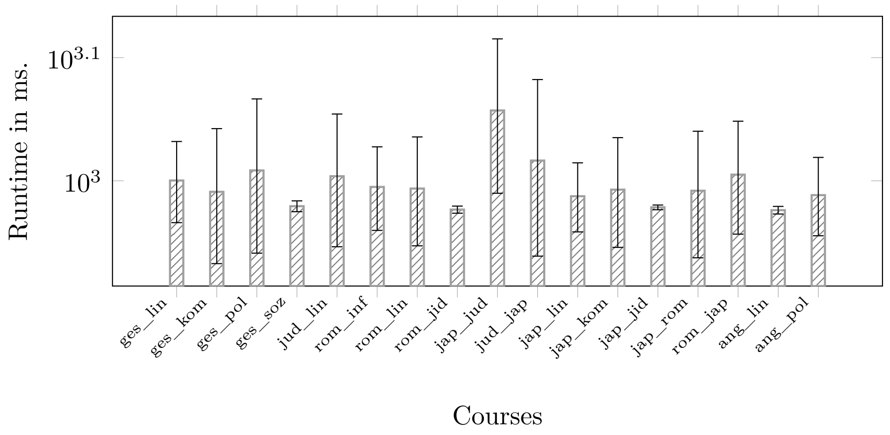


Figure 6.20: Z3 results for **infeasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*.

6.4.3 Alloy

The next formalism used to explore the timetable validation problem is Alloy [71], which we already briefly discussed in Chapter 3.

Alloy is a modelling language geared towards software validation and verification. The language is based on relational logic, combining relations and first order logic. Alloy models are created by defining signatures, that represent sets of values and relations linking them. Constraints are defined as facts and assertions about the signatures. Figure 6.21 shows an artificial example based on Figure 6.3. This example is composed of a singleton signature `A` that contains a `value` field of type `Int`. Associated to the signature is a fact that constraints valid instances to those where `value` is greater than 99.

```
one sig A {
  value : Int
} {
  value > 99
}
```

Figure 6.21: Example of an Alloy signature with one field and an associated fact constraining valid instances.

The Alloy language was designed from the outset to be translatable to SAT problems. This leads to certain restrictions in expressivity, e.g., higher-order relations and sets are not allowed as their SAT encoding would become too large to be tractable. Additionally, the Alloy Analyzer works based on a bounded universe of atoms. The same applies when handling integers, where it requires an upper bound for the size (in bits) to be used to represent numbers in the SAT translation.

The language is tightly coupled with the Alloy Analyzer, the tool used to develop and validate Alloy models. The Alloy Analyzer provides an IDE for Alloy, tools to evaluate the models and a graphical representation to inspect model instances. The Analyzer takes advantage of SAT-Solvers, but in difference to Z3, is itself not an SMT Solver.

Problems expressed in the Alloy language are translated to SAT instances which are evaluated by a standard SAT-solver. The mapping from Alloy to SAT and the search for satisfying instances is controlled by Kodkod [125]. Kodkod is a library that provides a constraint solver for first order logic and relations that is used by Alloy and several other tools.[4]

The Alloy Analyzer works based on a bounded universe with a limited number of atoms in order to conduct exhaustive search of the possible states [99]. This means, if Alloy Analyzer does not find a model for a set of constraints, it does not prove the general unsatisfiability of the constraints but might be due to the bounded universe.

Since Alloy is designed as a model finding tool there is no explicit support in the tool, for execution traces and state transitions as it is in the B Method, nevertheless. It is possible to express state transitions in Alloy using an ordering on the set of instances for a given model. Additionally, as Alloy and B are based on first-order relational logic, they share many concepts and modelling approaches.

As noted in Chapter 3, ProB has a backend [108] which translates B constraints into SAT using the same Kodkod library that Alloy employs.

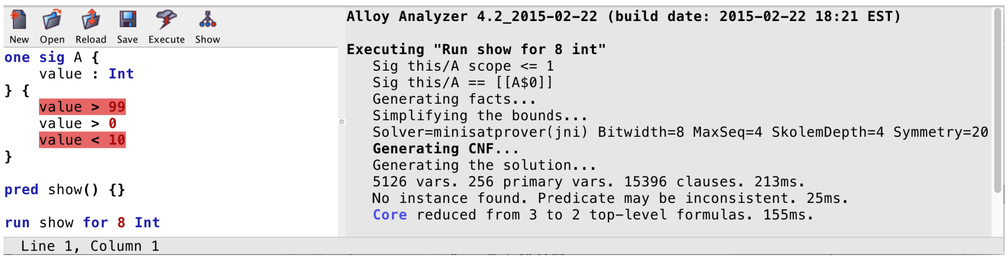
Unsatisfiable Core

Like Z3, the Alloy Analyzer provides built-in support for computing the unsatisfiable core of formulas where no model could be found [124]. The Analyzer considers the full specification, and not only marked assertions, and highlights formulas that are detected as part of the unsatisfiable core as shown in Figure 6.22, distinguishing constraints that are guaranteed to be part of the unsat core and constraints that are potentially, but not necessarily part of the unsat core. In Figure 6.22 we have extended the example from Figure 6.21 with two facts. The additional fact `value < 10` makes the facts unsatisfiable and is highlighted as part of the unsatisfiable core detected by the Analyzer.

Implementation

We represent the timetable data as an Alloy module. Each data type is represented by an abstract signature and each specific instance is represented by a singleton instance of a signature that extends the corresponding abstract signature. For instance, the different

[4]`http://emina.github.io/kodkod/` - [Online; accessed 28-March-2017]



```
one sig A {
    value : Int
} {
    value > 99
    value > 0
    value < 10
}

pred show() {}

run show for 8 Int
```

```
Line 1, Column 1
```

```
Alloy Analyzer 4.2_2015-02-22 (build date: 2015-02-22 18:21 EST)

Executing "Run show for 8 int"
   Sig this/A scope <= 1
   Sig this/A == [[A$0]]
   Generating facts...
   Simplifying the bounds...
   Solver=minisatprover(jni) Bitwidth=8 MaxSeq=4 SkolemDepth=4 Symmetry=20
   Generating CNF...
   Generating the solution...
   5126 vars. 256 primary vars. 15396 clauses. 213ms.
   No instance found. Predicate may be inconsistent. 25ms.
   Core reduced from 3 to 2 top-level formulas. 155ms.
```

Figure 6.22: Unsatisfiable core representation in the Alloy Analyzer.

```
enum Slot { a1,a2,a3,a4,a5, b1,b2,b3,b4,b5, c1,c2, ...}

enum Rhythm {weekly, biweekly_even, biweekly_odd}
```

Figure 6.23: Alloy representation of time slots and rhythms using `enum` signatures.

time slots are represented using Alloy's `enum` construct, shown in Figure 6.23, which creates a singleton set for each entry that extends the abstract signature `Slot`.

Sessions are represented as abstract signatures that contain a rhythm and a slot attribute. Each session in the data set is a singleton signature that extends this abstract signature, setting the corresponding attributes in a fact associated to each signature, see Figure 6.24 for an example.

Groups are represented similarly to sessions. Modules contain a set of the associated units, the number of elective units to be chosen in that module and two relations. One relation describing the type of each unit (mandatory or elective) and one to represent the semesters associated to each unit in that module. Courses, analogously, contain a set of modules, a relation describing the types of modules, if they are mandatory or elective in the course and finally the number of elective modules to be selected for the course.

The validation rules are represented using Alloy facts and evaluated using the `check` command. Course selection is done using the `checkFeasibility` predicate for a given course name, as shown in Figure 6.25. This predicate constrains valid instances of `Curriculum` to those where the course attribute is the given course *c*.

A solution for a given course is represented as a signature that contains fields for all properties of a solution, the relevant choices, e.g. of modules, are represented as functions.

```
abstract sig Session {
  rhythm: one Rhythm,
  slot: one Slot
}
one sig session1 extends Session {} {
  rhythm=weekly
  slot=a2
}
one sig session2 extends Session {} {
  rhythm=weekly
  slot=a3
}
```

Figure 6.24: Alloy representation of different **sessions**.

```
pred checkFeasibility(c : Course) {
  Curriculum.course = c
}
```

Figure 6.25: **Course** selection predicate used to set the course to be validated.

The `Curriculum` signature and one of the associated facts are show in Figure 6.26. The facts represent the constraints a valid timetable must satisfy, in particular the absence of binary conflicts. The `conflictFreedom` fact ensures, that for any pair of units, assigned to the same semester, there is no binary conflict among the selected groups.

If the Analyzer finds an instance of `Curriculum` that satisfies all constraints it represents a viable choice of classes to finish the degree as prescribed by the curriculum rules.

Enumeration Similar to Z3, the Alloy language is intended to declaratively define a problem and state the constraints that a solution has to satisfy, leaving the search for a variable assignment to the underlying SAT solver and then mapping the result to Alloy objects in case there is one. Due to the construction Alloy's use of external solvers the language provides no means to control how the underlying solver will search for a model.

```
one sig Curriculum {
  course: Course,
  modules: set Module,
  mandatoryModules: set Module,
  electiveModules: set Module,
  units: set Unit,
  mandatoryUnits: modules → set units,
  electiveUnitChoice: modules → set units,
  groupChoice: units → one Group,
  semesterChoice: units → one Semester
} {
  ...
}
fact conflictFreedom {
  some c : Curriculum •
    all a, b : c.units • (a ≠ b and c.semesterChoice[a] =
        c.semesterChoice[b])
      implies not unitsInConflict[a, b, c.groupChoice]
}
pred clash(s1, s2 : Session) {
  s1.slot = s2.slot and rhythmsInterfere[s1, s2]
}
pred rhythmsInterfere(s1, s2 : Session) {
  s1.rhythm = s2.rhythm or s1.rhythm = weekly or s2.rhythm = weekly
}
pred groupsInConflict(g1, g2 : Group) {
  all s1 : g1.sessions, s2 : g2.sessions • clash[s1, s2]
}
pred unitsInConflict(u1, u2: Unit, gc : Unit → lone Group) {
    let g1 = gc[u1], g2 = gc[u2] • groupsInConflict[g1, g2]
}
```

Figure 6.26: Curriculum signature and associated facts used to represent solutions.

Results

Unfortunately, the Alloy Analyzer is **unable** to cope with our evaluation data sets.[5] Although it remains unclear if it is a problem with the chosen representation, we assume that the size of the universe defined in the problem. For this large universe the translation as done by Kodkod yields a SAT representation that is too large. For small artificial examples used for testing and creating all implementations of the solver, the Analyzer is able to find models and conflicts. In our evaluation we used version `4.2_2015-02-22` of the Alloy Analyzer.

Related & Future Work

Alloy and the Alloy Analyzer have been used for data validation before, e.g. to validate security constraints for Java bytecode [110]. This approach works by translating Java bytecode on a method basis to Alloy, which combined with a template yields a valid model on which the JVM security constraints can be verified. Huynh performed a direct comparison of B/ProB and Alloy on the verification of a system for access control policies to medical data created at the Sherbrooke Hospital [68]. An approach relying solely on the constraint solving and model finding features of ProB did not yield any results, because ProB was not able to solve the constraints efficiently. An alternative approach combining constraint solving with operations and state variables to guide the process produced better results. In the evaluation, based on this second approach, ProB outperforms, by about two orders of magnitude, the Alloy Analyzer on the verification of randomly generated test instances of the models. As mentioned in Chapter 4, Yeung [130] explored the use of Kodkod to create a tool that generates a timetable with the classes needed to complete a given course.

Directly representing the problem as a Kodkod instance, without using Alloy as the input language permitting the use of features available in Kodkod but not exposed to Alloy, e.g. partial instances [101] seem to be a worthwhile alternative approach to this problem. Alternatively, creating instances with a smaller universe, by only adding to the specification the data relevant for one check and generating one specification for each course to be checked could yield better results.

[5]We cancelled a run trying to find a solution for the `ger_inf` course, which has a CI of 2, after 3 hours without result.

A direct encoding as a SAT problem should be faster than either Z3 or Alloy but exhibit similar problems, for instance coping with choices and subset selection, would be an interesting way of finding a lower bound for the performance of SAT based tools.

6.4.4 Python

The last language discussed here is Python [127]. Python is an object oriented dynamic language. Originally introduced in 1991 it is a popular language in areas ranging from web application development to scientific computing. The language is at the same time simple, powerful and extensible. We have chosen Python as an instance of an imperative, eagerly evaluated object oriented language.

One language feature we use in our solver are generators. Generators are functions, similar to coroutines, that can interrupt their execution at defined points and can be resumed later. Every time a generator yields control back to the invoking function it can pass a value to the caller. Generators are also similar and compatible to iterators in the iterator pattern [53], only that instead of using the state of the object to store the state of the iteration, generators use the frame of their function and its local variable.

Unsatisfiable Core

Python as a general purpose programming language has no native support for computing an unsat core of a predicate. Nevertheless it is possible to implement this as a recursive search, as done for our B model, where we collect all units for an arbitrary module combination (since all must contain a conflict if we assume the course is infeasible) and recursively try to minimize one of the possible sets of units for those modules. This has the advantage that we have control over the process of minimization and can manually decide which data elements we want to minimize and which we can ignore, but at the same time it is necessary to add that to the implementation upfront. Figure 6.27 shows, based on Figure 6.4, a Python implementation of an unsat-core search that returns one minimal unsatisfiable core from the passed list with regard to a predicate function.

```
def uc(predicate, lst, acc = None):
    if acc is None:
        acc = []
    if len(lst) == 0:
        return acc
    #
    item = lst[0]
    tail = lst[1:]
    if predicate(tail + acc):
        acc.append(item)
    return uc(predicate, tail, acc)

def pred(lst):
    return sum(lst) < 10

>>> uc(pred, [2,7,3,8,1,6])
[8,6]
...
```

Figure 6.27: Python implementation of a search for a minimal unsatisfiable core of a list of values with regard to a given predicate.

Implementation

In Python, we make use of objects and collections to represent the different types of entities in our data model. Each entity type is represented by a corresponding class, as shown in Figure 6.28 for **units**, e.g. a class for **courses**, **modules**, etc. Properties are represented as the attributes of the objects. Finally, each entity is an instance of the corresponding class, e.g. a course is an instance of the class `Course`.

The validation process in Python, as the others, follows pretty closely the schema described at the beginning of this chapter. In general the validation in Python is expressed as an exhaustive search through the tree of choices and combinations. For each configuration we collect the pairs of units. Each pair that is assigned, under the current configuration, to the same semester is checked for conflicts among the sessions of the selected groups, see `check_cs` in Figure 6.29. The check, whether a pair of sessions is compatible, is implemented by overloading the `&` operator in `Session.__and__`.

```
class Unit(Model):
    def __init__(self, idx, title, department, groups):
      self.idx = idx
      self.title = title
      self.department = department
      self.groups = groups
...

unit1 = Unit(1, "Unit 1", "Computer Science", [...])
```

Figure 6.28: Python class used to represent units.

Enumeration One aspect that is particularly challenging to represent in an imperative language is the nested enumeration of the variables used in such a problem. Enumeration variables are declared and used at different steps in the solver. Each variable needs to be enumerated to continue the search and instantiate other variables. The simplest approach to enumeration is to conduct a recursive search through the space of variables and values. The enumeration procedure picks a not yet enumerated variable, assigns the first possible value to it, and recursively calls itself. If the call returns without having found a solution, the enumeration would pick the next possible value for its selected variable and recursively call itself again, until either a solution is found or the possible values are exhausted. This effectively uses the recursion to create every possible path in the tree of variable valuations until it finds a path and a combination of values for the variables that satisfies the constraints. This approach has the disadvantage, that many runtime environments, such as the Java virtual machine as well as Python, limit the depth of the stack and thus the number of recursive calls allowed in the language to avoid an arbitrary growth of the execution stack. For this reason, we have chosen to implement the enumeration using generators. One aspect that is covered automatically by using recursion, but has to be handled manually when using iterators, is that of restarting the enumeration of different variables to ensure that all combinations are covered.

To address this we have created an enumeration module that is instantiated with a mapping of variables and their ranges. Each possible combination of values is represented as a map from variable name to its current value. The enumeration for each variable is represented as a generator that yields each of the possible values.

```
class Session(Model):
    ...
    def __and__(self, other):
        if (self.rhythm == 0 or other.rhythm == 0
            or self.rhythm == other.rhythm):
            return self.slot == other.slot
        return False
...

def check_cs(cs, group_choice, semester_choice):
    for u1, u2 in cs:
        if semester_choice[u1] != semester_choice[u2]:
            continue
        g1 = u1.groups[group_choice[u1]].sessions
        g2 = u2.groups[group_choice[u2]].sessions
        pairs = ((a,b) for a in g1 for b in g2)
        for (s1, s2) in pairs:
            if s1 & s2:
                return False
    return True
```

Figure 6.29: Python implementation of the conflict check for two units.

Generators are organized as a linked list, ordered by the size of each variable's domain. Each request for a new combination "pulls" the current state out of the chain of iterators and returns it as a combination. When we request one result all iterators are advanced such that we can return an assignment for each variable. The next value is produced by advancing the innermost iterator until it is exhausted. Once it is exhausted the preceding iterator is advanced and the innermost iterator is replaced by an enumeration of its cached results. This model is built on the assumption that the iterators do not depend on external state. Basically, this model moves the recursion and backtracking model to the heap and manages the recursion by exhausting and restarting iterators along the chain of iterators.

Table 6.6: Summary of results for Python based implementations.†

Tool	Results			Runtime in ms.	
	# Feasible	# Infeasible	# Timeouts	Average	Median
Arts & Humanities					
Python	43	13	11	313811.63	1.71
Python 3	42	13	12	361611.18	1.94
PyPy	43	13	11	290756.97	2.00
Business Administration & Economics					
Python	5	0	1	317525.38	1244.17
Python 3	3	0	3	901022.75	917463.25
PyPy	5	0	1	300686.84	19.46

† We have taken timeouts into account when computing the average and median for each tool. Each timeout represents a runtime of 30 minutes.

Results

The evaluation for Python was performed using three different implementations of the language: the official Python interpreter (CPython) in versions 2.7.12 and 3.5.2 and PyPy, using version 5.6.0 corresponding to language version 2.7.12.

The Python 2.x family is still maintained and widely used, but new features are only added to the 3.x versions. There are some syntactic differences between both versions of the Python language. Our model is written in a manner that is compatible with both versions, allowing us to compare them without having to modify our implementation.

The main difference among the implementations is the execution model. CPython (in both versions) is an interpreter that converts Python code to bytecode and interprets it. PyPy, on the other hand, contains a bytecode interpreter for the language, but additionally uses just-in-time (JIT) compilation to optimize frequently executed loops in a program [23]. The solver implementation in Python contains several nested loops, in addition to the loops used to provide the enumeration of variables, which is a reason to expect a speed-up from using a just-in-time compiler for frequently executed code paths.

As with the Prolog based brute-force approach, for the Python based benchmarks we excluded instances from the evaluation that had not returned a result after 30 minutes (exclusion based on the Python 2.7.12 reference implementation). On Python 3.5.2 we had to additionally exclude the AH course `rom_ang` as well as the BAE courses `wichem_bachelor` and `vwl_bachelor` that did not completed the check within the time frame.

A summary of the results for all three Python runtimes can be found in Table 6.6. The summary highlights the number of checks that completed or did not finish before the timeout limit. Table 6.7 shows a selection of the results for the three compared python runtimes, all collected results are presented in Appendix D.4. Visualizations for the results for the AH data set are presented in Figure 6.30 and Figure 6.31 organized by feasible and infeasible courses and in Figure 6.32 for the BAE data set.

The results show that all three versions behave mostly similar on both data sets. They also show that, as expected for a brute-force solution, detecting infeasible courses, where the solver needs to explore all possible instances is generally slower than checking feasible instances where a solution might be found before having explored all possible instances.

PyPy performs worse for smaller instances, which is to be expected, since PyPy's just-in-time compiler has a known overhead on start-up until it has detected hot loops, traced their execution and generated code for frequent paths. PyPy also performs worse for some larger feasible instances, where either a solution is found quickly and the overhead is to big or the search takes too many different paths, forcing the JIT to trace and compile many different paths. On instances with larger CI values PyPy performs, with a few exceptions, better than both CPython versions.

Python 3 shows the worst performance, with one additional timeout on the AH data set and two additional ones on the BAE data. Additionally, Python 3 is either on par with the Python 2 results or, sometimes even significantly, slower.

Table 6.7: Python results for the *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set.

Course	Feasible	Python 2		Python 3		PyPy		CI
		Runtime[‡]	SD	Runtime[‡]	SD	Runtime[‡]	SD	
				Business Administration & Economics				
wichem_master	✓	0.65	±0.09	4.60	±6.71	27.04	±1.54	8352
bwl_master	✓	0.83	±0.1	2523.54	±6856.63	10.61	±0.88	180153
vwl_master	✓	0.89	±0.017	3608.24	±11006.57	1.30	±0.11	217203
wichem_bachelor	✓	102599.04	±1435.42	timeout[†]	-	2.66	±0.22	32265043
vwl_bachelor	✓	2550.88	±65.42	timeout[†]	-	4079.46	±176.96	475079582
bwl_bachelor	timeout[†]	-	-	-	-	-	-	2067530678
				Arts & Humanities				
jap_jud	✗	39207.51	±682.05	41327.29	±7440.51	4881.58	±245.77	497664
jud_ang	✓	1.25	±0.17	1.32	±0.05	3343.34	±64.17	1244160
ger_jap	✓	7.81	±0.73	4.60	±1.90	298.92	±24.80	1990656
jap_ger	✓	7.94	±0.65	4.59	±1.96	348.43	±21.82	1990656
jap_lin	✗	73694.18	±476.73	79774.47	±10584.42	10291.55	±180.66	1990656
jap_kom	✗	337939.57	±5700.33	351540.90	±35868.44	38805.63	±578.21	4478976
jap_pol	✓	126165.56	±1667.92	203076.17	±73462.78	5769.71	±217.50	4478976
jap_soz	✓	30.54	±1.59	20.11	±11.36	2.19	±0.30	4478976
ger_ang	✓	1.05	±0.01	1.58	±0.27	0.54	±0.00	4976640
jap_jid	✗	237109.51	±1937.08	278604.28	±44212.45	34371.42	±481.77	7962624

[‡] Runtime in milliseconds.
[†] Each timeout represents a runtime of 30 minutes.

Table 6.7: Python results for *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set.

Course	Feasible	Python 2		Python 3		PyPy		CI
		Runtime‡	SD	Runtime‡	SD	Runtime‡	SD	
ang_ges	✓	`270981.06`	`±4494.46`	`1498607.53`	`±1784865.99`	`389613.35`	`±3139.16`	100776960
ang_inf	`timeout`†	-	-	-	-	-	-	201553920
jap_rom	`timeout`†	-	-	-	-	-	-	382205952
rom_jap	`timeout`†	-	-	-	-	-	-	382205952
ang_jud	✓	`2929.73`	`±60.31`	`9499.34`	`±12106.49`	`998072.90`	`±8646.66`	403107840
rom_ang	✓	`72734.07`	`±480.07`	`timeout`†	-	`229.60`	`±1.83`	955514880
ang_ger	✓	`18269.50`	`±150.57`	`77729.28`	`±100349.57`	`727.39`	`±48.53`	1612431360
ang_lin	`timeout`†	-	-	-	-	-	-	1612431360
ang_kom	`timeout`†	-	-	-	-	-	-	3627970560
ang_pol	`timeout`†	-	-	-	-	-	-	3627970560
ang_soz	`timeout`†	-	-	-	-	-	-	3627970560
ang_jid	`timeout`†	-	-	-	-	-	-	6449725440
jap_ang	`timeout`†	-	-	-	-	-	-	38698352640
ang_rom	`timeout`†	-	-	-	-	-	-	309586821120
ang_jap	`timeout`†	-	-	-	-	-	-	12538266255360

‡ Runtime in milliseconds.

† Each timeout represents a runtime of 30 minutes.

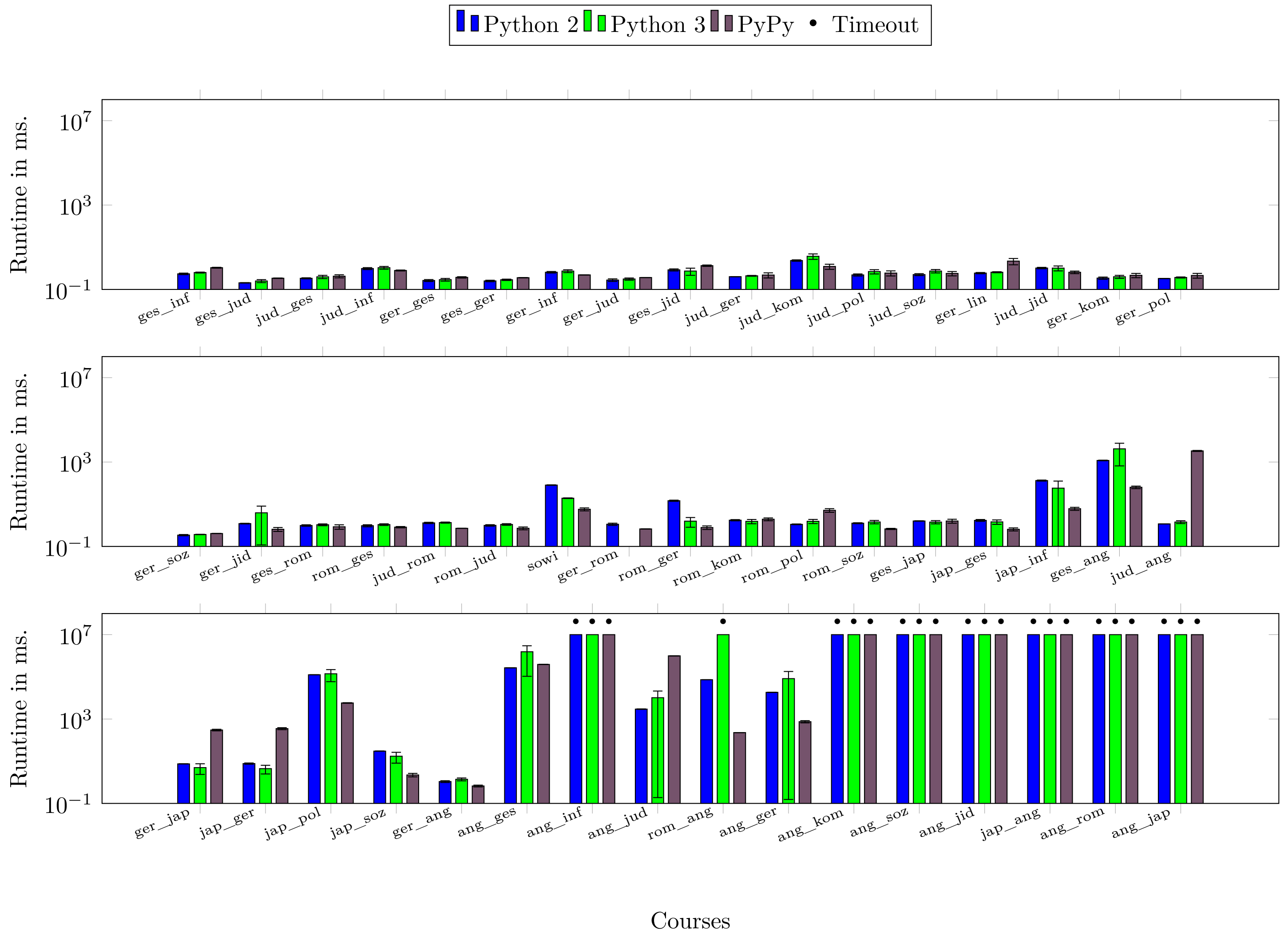


Figure 6.30: Python results for **feasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*.

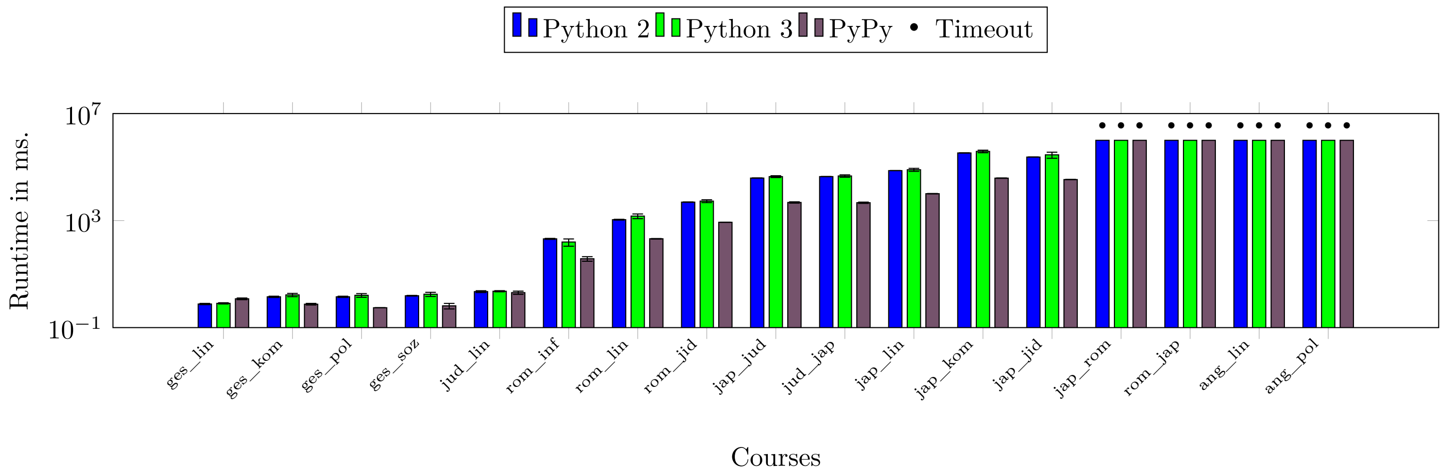


Figure 6.31: Python results for **infeasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*.

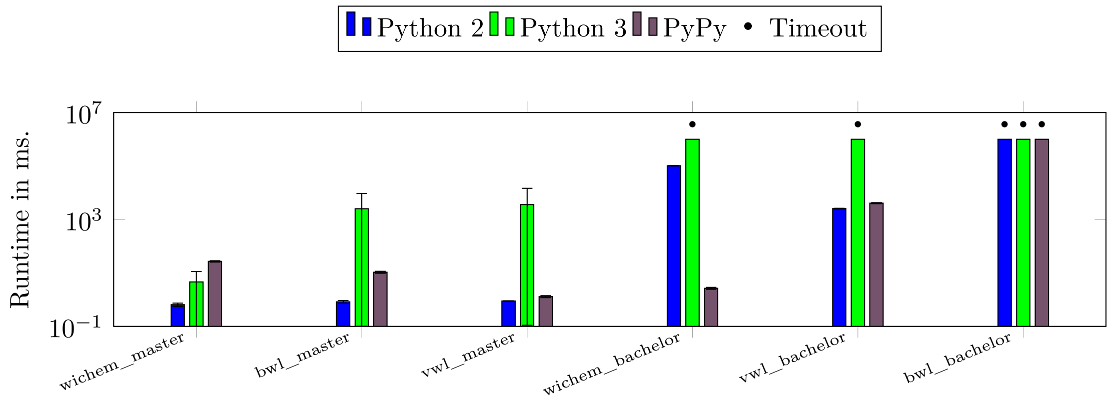


Figure 6.32: Python results for *Business Administration & Economics* data set ordered by increasing *complexity index*.

Table 6.8: Summary of results as computed by ProB.[†]

		Arts & Humanities	*Business Administration & Economics*
Results	# Feasible	50	6
	# Infeasible	17	0
	# Timeouts	0	0
RT[‡]	Average	52.37	211.43
	Median	50.00	185.00

[‡] Runtime in milliseconds.
[†] We have taken timeouts into account when computing the average and median for each tool. Each timeout represents a runtime of 30 minutes.

6.4.5 ProB/B

We have discussed our B and ProB based solution to this problem in Chapter 4. In order to compare the results for the different alternative implementations we present the results for running our benchmarks on the second version of our models (See Table 4.2 for details).

Version 3 For comparison we have included in Appendix D.5 a table showing the current results for the AH data set as computed using the third version of our tool.

Table 6.8 presents as summary of the results computed using ProB. All results for the BAE data set and results for the 25 larges CI values in the AH data set are presented in Table 6.9. All results collected with ProB are shown in Appendix D.5.

The results show that ProB is able to find a solution for every course in the AH data set in less than 160 milliseconds with a median runtime of 50 milliseconds. For the BAE data set the longest runtime is of 387.14 milliseconds with a median of 185 milliseconds. Visualizations for the AH data set be found in Figure 6.34 and Figure 6.35 grouped by feasible and infeasible courses and in Figure 6.33 for the BAE data set.

For ProB `ang_jap`, the course with the largest CI value, leads to the longest computation time of 157 milliseconds to decide the feasibility of the course. The same applies to the BAE data set, where `bwl_bachelor`, the instance with the largest CI value, has an average runtime of 387.14 milliseconds. There is a certain overhead associated with

Table 6.9: ProB results for the *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set.

Course	Feasible	Runtime in ms.	SD	CI
		Business Administration & Economics		
wichem_master	✓	97.14	±7.56	8352
bwl_master	✓	124.29	±7.87	180153
vwl_master	✓	101.43	±3.78	217203
wichem_bachelor	✓	238.57	±6.90	32265043
vwl_bachelor	✓	320.00	±5.77	475079582
bwl_bachelor	✓	387.14	±28.12	2067530678
		Arts & Humanities		
jap_jud	✗	105	±5.27	497664
jud_ang	✓	51	±5.68	1244160
ger_jap	✓	64	±6.99	1990656
jap_ger	✓	67	±8.23	1990656
jap_lin	✗	106	±14.30	1990656
jap_kom	✗	46	±6.99	4478976
jap_pol	✓	73	±4.83	4478976
jap_soz	✓	68	±6.32	4478976
ger_ang	✓	46	±6.99	4976640
jap_jid	✗	67	±8.23	7962624
ang_ges	✓	64	±5.16	100776960
ang_inf	✓	99	±5.68	201553920
jap_rom	✗	103	±6.75	382205952
rom_jap	✗	106	±6.99	382205952
ang_jud	✓	62	±6.32	403107840
rom_ang	✓	119	±9.94	955514880
ang_ger	✓	71	±5.68	1612431360
ang_lin	✗	61	±3.16	1612431360
ang_kom	✓	78	±7.89	3627970560
ang_pol	✗	59	±5.68	3627970560
ang_soz	✓	79	±7.38	3627970560
ang_jid	✓	102	±9.189366	6449725440
jap_ang	✓	119	±7.378648	38698352640
ang_rom	✓	148	±7.888106	309586821120
ang_jap	✓	157	±13.374935	12538266255360

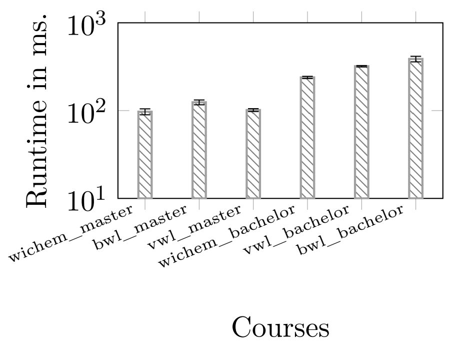


Figure 6.33: ProB results for courses in the *Business Administration & Economics* data set ordered by increasing *complexity index*.

the execution in ProB, even very simple cases take at least 20 milliseconds on average. Part of this overhead is, as mentioned in Chapter 3, associated to the JIT compiler added to SICStus Prolog in release 4.3. Nonetheless, the largest instance on the AH data set takes only 7.85 times longer, on average, than the smallest, while the CI value is 62691331276801 times larger. On the BAE data set the biggest CI value is 247549 times larger and the runtime is only four times longer.

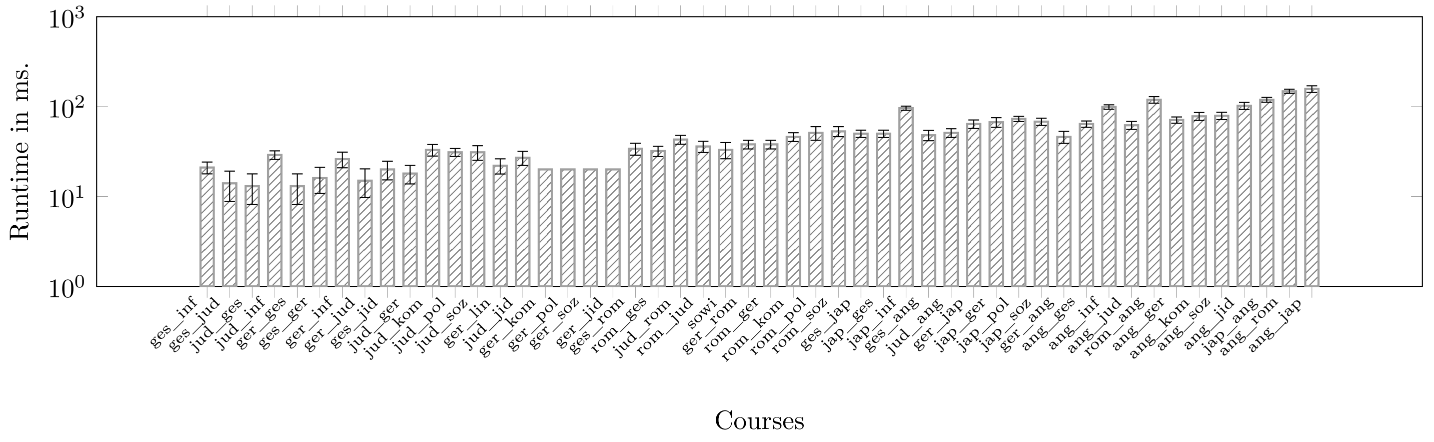


Figure 6.34: ProB results for **feasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*.

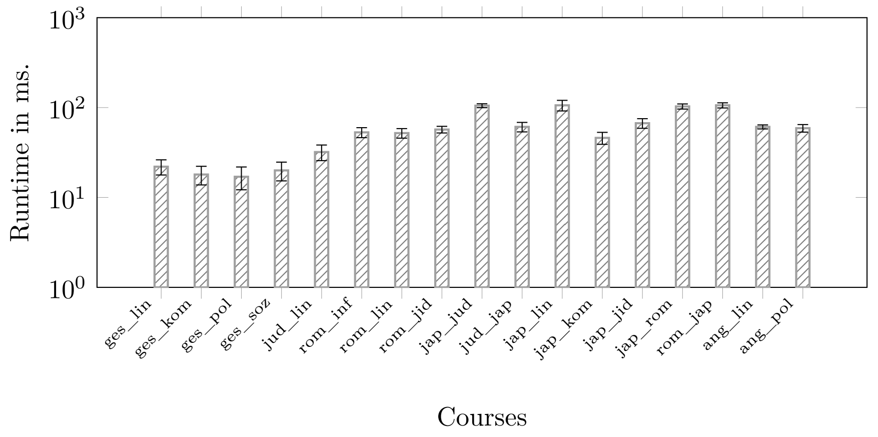


Figure 6.35: ProB results for **infeasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*.

Table 6.10: Filesize of models (solver and generated data) in bytes for each faculty and language/tool.

Language/Tool	Data	Solver	Total
Business Administration & Economics			
Alloy	190668	2899	193567
B	151829	36532	188361
Prolog	149689	12494	162183
Prolog clp(FD)	149689	7251	156940
Python	264942	17598	409370
SMT-LIB	276720		276720
Arts & Humanities			
Alloy	216725	2899	219624
B	143500	36532	180032
Prolog	126893	12494	139387
Prolog clp(FD)	126893	7251	134144
Python	135406	17598	462514
SMT-LIB	601865		601865

6.5 Discussion

In this section we will compare the different tools and implementation with regard to the areas we highlighted in the discussion of each. We will compare how the enumeration and model search in the different tools works, how these languages support the conflict source detection, how quickly they can detect conflicts in our data sets and how well the languages are suited for these kinds of validation problems.

6.5.1 Enumeration

One of the big differences among the different approaches is how the enumeration of possible solutions to be tested for feasibility works. Having to nest the enumeration with the problem description makes the encoding rather cumbersome. This is one aspect where modelling and declarative programming languages are better suited for encoding this kind of problem, since they clearly separate the encoding from the search strategy. Nevertheless, for declarative approaches how a problem is represented still can have an effect on how the search is performed.

High-level approaches geared towards validation, model checking and proof such as B, Alloy and Z3 hide how the tools enumerate values from the user to a higher degree than low-level tools. Although B has loops as part of its substitution language, which we have not used here, the mathematical language of B has no concept of explicit enumeration at the language level, and relies on quantifications to express this.

Removing control over the enumeration from the model has the advantage of clearly separating the modelling from the search for satisfying instances. At the same time this can be a disadvantage if the chosen enumeration strategy does not yield the expected results. This means that in cases where the heuristic behind the enumeration does not work, the possibilities to try alternate approaches are limited. Nevertheless, the tool itself has greater control over the enumeration and can choose different strategies based on the input, that might yield better results than a poorly configured search. Prolog/clp(FD), for instance, permits the user to choose a strategy how variables are labelled. To illustrate this on our solver, for the AH course `ang_rom` the runtime of the validation is 10 milliseconds when using the `ffc` enumeration strategy, which enumerates the most constrained variable first. This runtime increases to 12 seconds when changing the enumeration strategy to `max`, enumerating the variable with the largest upper bound first.

The imperative solution requires explicitly considering how to collect and when to enumerate values. It is also necessary to create an abstraction to represent enumeration variables, since there is usually no concept of a free variable that can be passed around. This can be solved either through backtracking and re-assigning a value or by adding an indirection. An indirection can be expressed using for instance an object that encapsulates the value and possibly additional information, e.g. the domain of the variable, or as in our Python implementation, using a map from variable name to current value. In Prolog, on the other hand, logical variables are part of the language that can be stored and passed around without unifying them with a ground value, thus allowing them to be unified at a later point in the execution.

SMT-LIB, since it does not support higher-order data types leads to an encoding that mixes the data representation with the validation rules. This basically requires unrolling parts of the model that could be represented using lookups into nested structures in other formalisms.

Table 6.11: Filesize in bytes of templates used to generate data models for each language/tool.

Language/Tool	Template
Alloy	3814
B	4699
Prolog	2286
Python	2596
SMT-LIB	5786

6.5.2 Unsatisfiable Core

Computing the source of a conflict is also an area that is very different among the tools. Z3 and Alloy support this natively. Prolog and Python, on the other hand, as general purpose languages, require the user to implement it for the domain, similarly to what we have shown in the respective sections (Figure 6.4 and Figure 6.27). The B language itself has no concept of an unsat core either. ProB has supported minimizing unsatisfiable `PROPERTIES` for quite some time and it has lately been extended to provide, as part of the language, restricted support for computing the unsatisfiable core of a predicate in form of an external function. The function takes a set to be minimized and a lambda used to decide if values from the set are part of the unsatisfiable core or not. The way this is handled in ProB (a similar approach to the one described for Prolog and Python) has the advantage, that it is possible to control the process more directly, only reducing relevant data sets instead of all elements in a predicate. A similar result can be achieved with Z3, since it only considers named assertions when computing an unsatisfiable core, which can mimic the behaviour describe above, but requires the user to label all relevant assertions. Alloy works fully automatic and detects all parts of a model that are associated to an unsat core, allowing the user to trace the conflict through the model.

One useful aspect of computing the unsat core using an external function, is that the return value of the function is available within the language. This means the value returned can be stored or used as part of another computation. In contrast to this the minimization of `PROPERTIES`, the unsat core in Z3 or in the Alloy Analyzer are only available as outputs or from within the respective tools or APIs.

Table 6.12: Summary of benchmark results.[†]

Tool	Results			Runtime in ms.	
	# Feasible	# Infeasible	# Timeouts	Average	Median
			Arts & Humanities		
Prolog	42	13	12	362,922.19	10.00
Prolog clp(FD)	50	17	0	1,603.79	0.00
Z3	50	17	0	17,126.43	20,035.00
Python	43	13	11	313,811.63	1.71
Python 3	42	13	12	361,611.18	1.94
PyPy	43	13	11	290,756.97	2.00
ProB	50	17	0	52.37	50.00
			Business Administration & Economics		
Prolog	3	0	3	900,228.50	900,700.00
Prolog clp(FD)	6	0	0	214.33	0.00
Z3	6	0	0	97,903.66	95,995.00
Python	5	0	1	317,525.38	1,244.17
Python 3	3	0	3	901,022.75	917,463.25
PyPy	5	0	1	300,686.84	19.46
ProB	6	0	0	211.43	185.00

[†] We have taken timeouts into account when computing the average and median for each tool. Each timeout represents a runtime of 30 minutes.

6.5.3 Performance

We have created diagrams for all benchmarked tools comparing them with ProB. All runtimes have been normalized to ProB. As in previous sections Figure 6.37 shows the relative runtimes for feasible courses in the AH data set, Figure 6.38 shows the infeasible ones and Figure 6.36 shows the results for the BAE data set. In cases the computed average runtime for an implementation is 0 the bar is missing and annotated with a ◦ symbol in the diagrams. Bars annotated with a • symbol represent timeouts. Additionally, in Table 6.12 we present an overview of all collected results. For each tool and data set it shows the number of feasible and infeasible courses that were validated and the number of computations that timed out. For each tool and data set we also present the average and median runtimes, both taking timeouts into account.

The benchmark results show that the brute-force approaches, either in Python or Prolog work well for small instances and for larger instances that are feasible. At a certain point they cannot compete with other approaches to this problem when validating instances with large CI values, in particular infeasible ones. Also the brute-force approaches and Alloy are the only ones that time out for large CI values.

Of the model/constraint based approaches that are able to solve all instances Z3 yields the worst results, being slower than clp(FD) and ProB for all instances considered. Nevertheless, Z3 is very consistent in its behaviour, even when checking instances with very large CI values. Z3 detects infeasible instances in about one second and decides if an instance is feasible in up to 35 seconds. This is still not on par with SICStus Prolog or ProB, but works for all instances we have used in this chapter. One of the problems we faced with Z3, that is possibly a reason why it is considerably slower than other approaches, is the difficulty of expressing cardinality constraints, i.e. selecting exactly n elements from a set. We have expressed this using a function that maps the set's elements to 0 or 1 and placing a constraint on the sum of the function values. We require the sum of the values in the function's range to be n, hence the values that map to 1 build a set of exactly n elements. As mentioned before, initial experiments using bitvectors to represent sets and counting bits to express cardinality constraints did not improve the runtime for infeasible instances and worsened the results for feasible instances.

Which approach works better, either the clp(FD) based ProB or Z3, strongly depends on the kind of problem being evaluated. In this case we have translated a problem that we had successfully validated using ProB to SMT-LIB. When evaluating our SMT-LIB model with Z3 it was slower than ProB on all evaluated instances. Krings, in his doctoral thesis [79, Chapter 5], created a translation from SMT-LIB to B. The translation was created with to goal of gaining access to problems and benchmarks collected in the SMT-LIB benchmark repository[6] and to use them to evaluate ProB's constraint solving features. For those tests solved by all considered solvers, ProB was significantly slower than either Z3 or CVC4 for feasible and infeasible instances. Generally, of those tests solved by ProB, it was on average slower detecting unsatisfiable instances than satisfiable ones, but was able to detect inconsistencies on instances that neither of the other compared tools could solve. Bride et al. [25] compared the use of SMT and

[6] `https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks` - [Online; accessed 25-September-2017]

CLP based techniques for the verification of workflow nets using Z3 and SICStus Prolog respectively. In their results the SMT based approach performs better than the CLP based one except for one type of workflow net.

Alloy unfortunately did not work on our benchmark data but only on test data. We suspect one of the reasons was the overhead of translating the input data to a SAT representation. As previously discussed in this chapter, in the case study conducted by Huynh, a purely constraint based approach to describe access control policies to medical data in B could not successfully be evaluated using ProB. An alternative approach combining constraint solving with the animation and model checking features of ProB outperformed an analogous Alloy model by about two orders of magnitude.

SICStus Prolog using clp(FD), even in the restricted manner we do in this evaluation, provides the best results in most cases. clp(FD) scales well and is able to solve all instances, solving, with three exceptions, all instances in less than one second. In particular the low overhead associated with directly encoding the problem in Prolog yields results for most instances very quickly, with a median runtime of zero milliseconds (considering SICStus' time resolution of 10 milliseconds).

Lastly ProB, which partly relies on clp(FD) internally is able to check all instances in this evaluation in up to 387 milliseconds for the BAE data set and 157 milliseconds for the AH data set. ProB has a higher overhead, which is particularly noticeable for small instances when compared to the brute-force and clp(FD) approaches. The overhead can be attributed to B's abstraction level and the preprocessing ProB has to do in order to evaluate a formula. The additional cost for small instances is contrasted by the capacity to quickly find results even for very large instances. In a few cases ProB is faster than Prolog/clp(FD) finding a result, notoriously for the course with the largest CI (`ang_jap`) where our Prolog/clp(FD) based solver takes 104 seconds and ProB only takes 157 milliseconds to decide that the course is feasible.

6.5.4 Complexity

The complexity of each implementation strongly depends on the features provided by each tool and exposed in the corresponding language.

Due to its lack of high-level data structures, formalizing this problem using Z3, makes it necessary to unroll certain parts of the search when generating the model. This creates several assertions, that state explicitly what could be represented using higher-order data structures in other formalisms. As an example: Figure 6.17 shows how we represented in SMT-LIB the rule that if unit uid_1 is selected, only one of the groups $gid_1..gid_6$ is allowed to be selected and none if the unit is not selected. In the model the assertion from Figure 6.17 is repeated similarly for each unit in the data set instead of expressing the same by traversing an analysing a data structure. In the other languages used in this chapter representing such a configuration constraining the search is part of the model and contained in the data structures instead of having to be made explicit.

Alloy is rather well structured, but is not always intuitive to use. The use of signatures is very helpful in creating well-structured and easy to follow specifications. As mentioned before, the reusable predicates and functions available in the language are very helpful vehicles of abstraction and reuse.

For the Python encoding it is necessary to make all steps explicit, in particular the enumeration. This leads to a close coupling with the validation of candidate instances. The effect it has on the implementation makes the way the validation is done in Python harder to use and follow than a declarative approach.

Lastly, both Prolog based implementations take advantage of the built-in concepts of recursion, backtracking and unification to express the problem. Prolog's execution model, where it is possible represent the search and enumeration, without having to make it explicit is a good match for this kind of validation problem. The extension of our solver with clp(FD) was rather simple, since the concept integrates well with Prolog. Using coroutines and constraint programming added additional complexity to the model by creating coroutines and by moving the enumeration of most values to the labeling step. Still this change allowed us to separate those values that are enumerated as part of the problem setup (which would be part of the generated model in SMT-LIB) and those that are enumerated as part of the search for a satisfying model.

Table 6.10 reports the filesize in bytes of each model. For each model we show the size of the generated code used to represent the data and the size of the model representing the validation rules. As mentioned before, in the case of the SMT-LIB encoding it was not possible to separate the logic from the data, hence we report only one number representing the combined data and logic descriptions. To put these numbers in relation, the size of the template which contains the logic and only the patterns for the data has 6090 bytes. The filesizes of the different templates used to generate the models are reported in Table 6.11. The size of the generated data models depends on the number of entities in each data set, which affects in particular the SMT-LIB model where we have to unroll more aspects of the data set when generating the model.

The largest generated data models correspond to Python and Z3. Python as a language is more verbose than the declarative languages discussed in this chapter. One aspect that accounts for most of the size of Python's data model is code that is generated to setup the relations between the different entities when the solver is started. This is particularly noticeable on the BAE data model, that contains more links between courses, modules and units than the AH data set. In the case of Z3 the size is largely due to the unrolling, where rules and data have to be combined and added explicitly to the generated model. One of the reasons the B solver implementation is larger than the others, is the fact that our B model supports more features than those considered for this evaluation. The model provides an interface to the main application using B operations and contains operations for computing conflicts and alternative time slots. A further reason is that being the basis of the application we have built, more work has gone into the modelling and into making sure the tool works as expected with ProB.

A further aspect we have not considered in this evaluation, is how to embed the different tools in an application. In Chapter 4 we explained how we have created an application around our B model. The application exposes the features of the model as HTTP based server application that passes request to ProB using a Java API. ProB is embedded in the application and used to evaluate the validation requests. Python and Prolog as general purpose programming languages, provide the necessary bindings to create a server or GUI application that embeds the solver. To create an application around Z3, the Alloy Analyzer or ProB these need to embedded in the application. The Alloy Analyzer provides a Java API that can be used to embed it in an application and interact with the model.[7] A further alternative would be to directly interact with the Kodkod library.[8] Z3 provides bindings for several languages, among them Java and Python.[9] ProB, as previously discussed, also provides a Java based API [16] that can be used for embedding.

[7]`http://alloy.mit.edu/alloy/documentation/alloy-api/` - [Online; accessed 17-April-2017]
[8]`http://emina.github.io/kodkod/release/current/doc/` - [Online; accessed 17-April-2017]
[9]`https://github.com/Z3Prover/z3#z3-bindings` - [Online; accessed 30-March-2017]

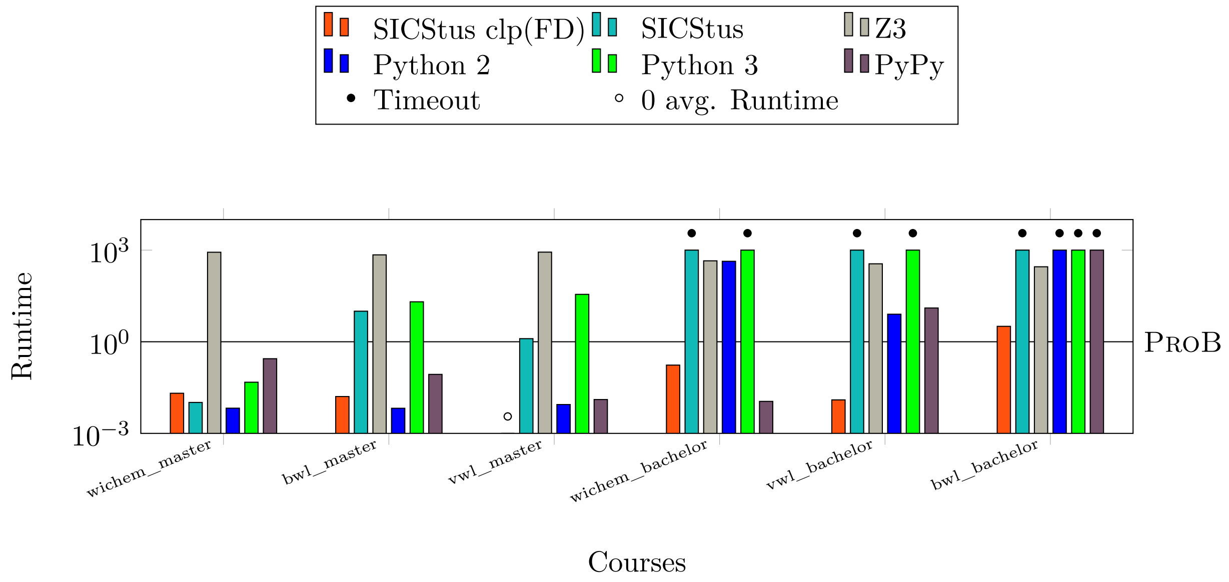


Figure 6.36: Relative runtimes normalized to ProB on the *Business Administration & Economics* data set. Courses displayed in ascending order by *complexity index*.

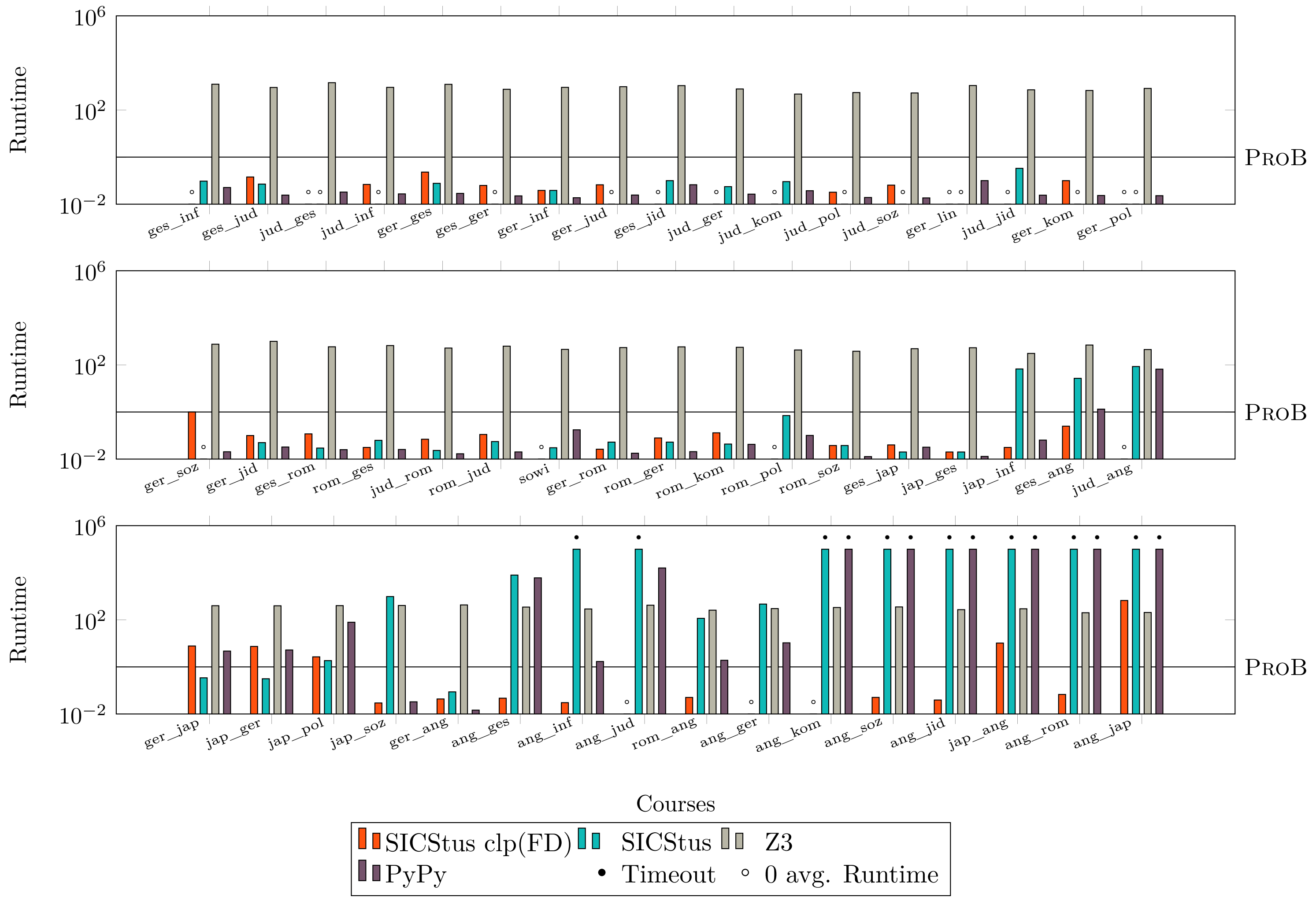


Figure 6.37: Relative runtimes for **feasible** courses on the *Arts & Humanities* data set normalized to ProB and shown on a logarithmic scale.

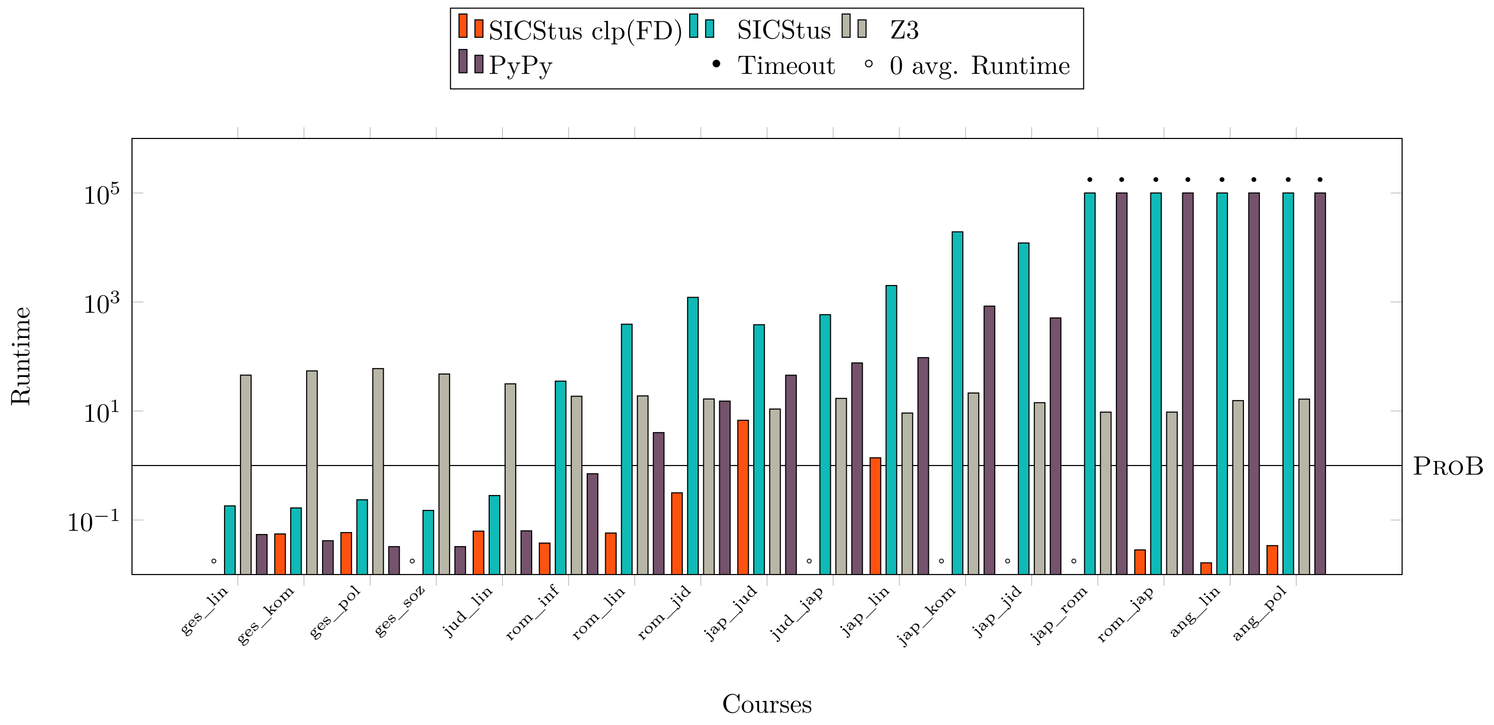


Figure 6.38: Relative runtimes for **infeasible** courses on the *Arts & Humanities* data set normalized to ProB and shown on a logarithmic scale.

6.5.5 Conclusion

In this chapter we have presented a simplified version of our case study from Chapter 4. Based on it we have discussed implementations in SICStus Prolog, using a brute-force and a clp(FD) based approach, Python, SMT-LIB/Z3, Alloy and B/ProB. We have evaluated each tool and language with regard to what was needed to implement our case study and with regard to their capability and performance to solve the validation problems on the provided data sets.

In general, the combination of B and ProB compares rather favourably to the tools we selected for this evaluation, which is reflected in the median runtime for ProB 50 milliseconds on the AH data set and of 185 milliseconds on the BAE data set. With regard to performance ProB provides good results, considering the high abstraction level of the B language. These results for ProB attest, compared to clp(FD) and brute-force approaches, a certain overhead when checking instances with small CI values. On the other hand the results are very good when checking instances with large CI values. On these instances ProB is faster than all compared tools besides clp(FD). Still, on a some instances ProB is even faster than the low-level clp(FD) encoding.

Zhou et al. [132] conducted a comparative study about the use of CP, IP, and SAT solvers from Prolog through a common interface. They concluded that none of the considered solvers is generally better than the others and that each approach is better suited for different kinds of problems. They summarize that constraint programming is well suited for problems with symmetry and global constraints, integer programming works well for problems that can be modelled by means of inequalities and SAT approaches work well for boolean problems. Our results, and those collected by Krings lead to a similar conclusion. Based on these results, ProB is well suited for problems on finite sets, integers and global constraints but might not perform well on very low-level problems.

As mentioned before, the fact that the B model is larger than the other encodings is partly due to its support for more features than the other implementations discussed here and the fact that more work has gone into the modelling and into making sure the tool works as expected with ProB and with the application built on top of it. These results have confirmed that it is possible to use a high-level modelling language to model and solve a constraint based problem and create an application on top of it that includes the discussed model as part of the executed program. The modelling process helped to improve the ProB constraint solver. The case study helped us to uncover bugs and performance problems in ProB.

This problem is very well suited to be modelled with the mathematical language of B. The validation rules can be expressed as B predicates that are evaluated on a machine's current state. The initial state of the machine is setup by evaluating the `PROPERTIES` of the machine, which are also used to derive information from the input data. At the same time the full B language contains many aspects, as discussed in the previous chapter, that do not fit well into the use of B in this scenario. An example of this is the lack of proper abstraction and composition mechanisms besides `DEFINITIONS` when relying on the predicate and expression language. Another example is the lack of `LET` and if-then-else in expressions and predicates. Admittedly using B for constraint modelling was not part of its originally intended use and leads to certain shortcomings in the use of the language when applied in this area.

The declarative approach has worked best for the problem discussed in this evaluation, in particular the separation of problem description from the search for solutions simplified the encoding and makes it easier to maintain and change. Although the loss of control over the enumeration process can lead to sub-optimal results. Knowing the structure of the problem would permit to take advantage of the structure during enumeration. While this would require an even more coupled implementation the declarative approaches have yielded the best results, in particular for problem instances with large CI values.

From the languages discussed here B and Alloy, despite all their shortcomings, are those best suited for this kind of validation problems. Both languages are based on relations and set theory. Alloy's signatures provide additional concepts, that are mapped to relations. Alloy might be even slightly better suited, since it is intended explicitly for model finding. Also the syntactic extensions on top of relations, such as signatures present a way of modelling concepts and relationships that is ideal for this scenario. Additionally the concept of named predicates, like named functions is a very powerful abstraction mechanism for a data validation problem. Unfortunately the Alloy Analyzer was not able to cope with the data sets used for this evaluation.

Several things would be interesting to explore in this regard: One is to use the Alloy language as an input for ProB for data validation problems; another would be to extend the B language with useful concepts for data validation such as named predicates, or creating, as with the rule validation language from Section 5.7.4 a superset of B that is better suited to encoding these problems. A different approach would be explore a tighter integration of ProB with other solvers, such as Kodkod, SAT or SMT solvers (experiments in this regard have been started by members of our research group).

7 Conclusion and Future Work

In this thesis we explored and discussed the use and usability of B and ProB as constraint modelling and solving tools. Our discussion has been guided by the following questions and goals as stated in Chapter 1:

(i) Evaluate the use of B not only as a formal modelling language but also as a constraint modelling language.

(ii) Evaluate if ProB can be used to efficiently find solutions to complex constraint problems modelled in B.

(iii) Explore if ProB can be used as a runtime for constraint based models and embedded in applications.

(iv) Analyse how the combination of B and ProB compares to other tools that can be used to model constraint based problems.

In Chapter 3 we discussed, that B can not only be used as a language for formal modelling and verification, but that the language is also very well suited to model constraint based problems. In particular the expression and predicate subsets of the language make it possible to succinctly express constraints declaratively at a high abstraction level. As we showed on the *jobs puzzle* it is even possible to stay very close to the natural language specification of a problem when modelling it using B. One expected advantage of the high abstraction level is that it makes it easier to validate a model with regard to the requirements and to communicate with domain experts, versed in mathematics but not necessarily in programming or modelling languages, about the representation of a problem. In the same chapter we demonstrated on the *jobs puzzle* and on several other puzzles that, besides being a vehicle to nicely model problems, these can be efficiently solved using ProB.

Based on these results, in Chapter 4 we conducted a larger case study about the applicability of the B and ProB to application development. We explored this in the context of university timetable validation. In cooperation with two faculties at our university we collected the different rules and constraints required to validate timetables. We formalized the collected rules as a domain theory of university timetables and created a B model based on them. Based on the model, we were able to create an application that uses this model at runtime and permits its users to validate and improve the curricula offered at the university. The application embeds ProB and the model and communicates with ProB via its Java API.

Based on the experiences gathered while working on this case study and on an independent project, in Chapter 5 we extract a common structure for data validation projects built using B on top of ProB. In Chapter 5 we described different approaches taken, as well as areas where we consider the language to be lacking certain features for this use and evaluated different alternatives how to improve and extend the language for data validation projects.

Finally, in Chapter 6 we conducted an empirical evaluation comparing different languages and tools on a simplified version of our case study. The goal was to assess how B and ProB compare to approaches that can be used for constraint modelling. We compared our solution to general purpose programming languages, constraint programming extensions to these and dedicated modelling languages. The evaluation is based on each tool's evaluation model, the complexity of implementing the case study in each language and on the runtime performance each tool showed when solving our case study. In general the B based approach occupies a middle ground, where the high-level modelling permits us to effectively model the timetable validation problem in a declarative manner. A high-level declarative model is, for this kind of constraint problems, a representation that is simpler and more maintainable than lower level approaches, either general purpose programming languages such as Prolog or Python or also than our SMT-LIB model. Nevertheless, a language such as Alloy targeted explicitly at model finding and structure description might be slightly more appropriate for this kind of modelling. On the other hand, regarding to the performance of the tools in our case study, ProB is, with a few exceptions, slower than a low-level encoding using Prolog and clp(FD). On small instances ProB shows a certain overhead, where even brute-force approaches are more efficient. In contrast, on large instances the constraint based approach of ProB shows very good results on this particular problem, where it is faster than brute-force approaches and Z3. But, as could be expected, the low-level approach using Prolog and clp(FD) is faster than ProB, with a few exceptions.

With all this we try to revisit the initial questions posed: Is it possible to express non-trivial constraint satisfaction problems in B and is it possible to use such a formal model at runtime for problem solving? Are language and the tools powerful enough to enable this usage scenario?

In general we can conclude, that B as a language is well suited to be used, not only as a formal specification language, but also as a constraint modelling language. In particular the mathematical subset of the language allows us to capture constraints very elegantly. As we showed in Chapter 3, it is in some cases even possible to stay close to the natural language representation of the problem.

B, as a specification language that was not originally designed as a constraint modelling language, has unsurprisingly several aspects that make its use as a constraint modelling language cumbersome. The extensions to the B language discussed in Chapter 5 showed possible approaches to extending and improving the language with regard to data validation and constraint modelling. Approaches could be creating either:

- Custom input languages for constraint modelling based on the mathematical notation.
- Domain specific extensions to the language, as done by Hansen and discussed in Chapter 5, that hide certain part of the complexity while making it easier to model specific problems.
- Front ends in ProB for existing modelling languages such as SMT-LIB as done by Krings, or Alloy where the language is closer to B but is specifically created as a modelling language.

A further approach would be to embed parts of the B language in a host language, in a way that naturally extends the host language's semantics. We performed initial experiments in this area by embedding the expression and predicate subsets of the B language into the Clojure programming language [64]. We provide a Clojure representation for this subset of the B language, such that the Clojure S-expressions are mapped to the corresponding B structures. The evaluation works by sending the mapped S-expressions to ProB for evaluation and mapping the results are back to clojure structures.

These are areas where we want to build upon the work presented in this thesis and explore different approaches to integrate the ProB constraint in different programming and modelling approaches. This could be achieved either by embedding it into a programming language, as described above for Clojure, or by offering additional front ends to the ProB constraint solver. These additional front ends can be either for general purpose modelling languages that map to B, domain specific languages or domain specific extensions to B.

Part IV

Appendices

A Publications

The manuscripts of the following publications I have co-authored have been used in this thesis. All of these publications where created in collaborative work, the contributions listed below are parts of these articles that are based largely on my work. Nevertheless, none of this would have been created without the contributions, discussions, proof-reading and editing by co-authors and colleagues.

A.1 Towards B as a High-Level Constraint Modelling Language

M. Leuschel and D. Schneider. Towards B as a High-Level Constraint Modelling Language - Solving The Jobs Puzzle Challenge. In Y. A. Ameur and K.-D. Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science (LNCS)*, pages 101–116, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

Contributions:

- Formalization and description of the of the *Jobs Puzzle*.
- Discussion of related work to the *Jobs Puzzle* model.
- Discussion of the subset sum, n-Queens and graph isomorphism problems. The respective implementations of the discussed solutions were created by Michael Leuschel and Daniel Plagge.
- Benchmarking of the n-Queens and peaceable army of queens implementations.

A.2 Model-Based Problem Solving for University Timetable Validation and Improvement

D. Schneider, M. Leuschel, and T. Witt. Model-Based Problem Solving for University Timetable Validation and Improvement. In N. Bjørner and F. S. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science (LNCS)*, pages 487–495, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

Chapter 4 is based on an extended version of this article. This extended version has been submitted to the Journal *Formal Aspects of Computing* to be considered for inclusion in a special issue about the *20th International Symposium on Formal Methods (FM 2015).*

Contributions:

- Development of a domain theory of timetables and curricula.
- Description of the formal model used for timetable validation. The model was created in cooperation with Michael Leuschel.
- Discussion of the related work on timetabling.
- Implementation of the application that embeds the formal model to validate timetables. The implementation of this application was created in joint work with Tobias Witt, Philip Höfges and Joshua Schmidt.

A.3 Using B and ProB for Data Validation Projects

D. Hansen, D. Schneider, and M. Leuschel. Using B and ProB for Data Validation Projects. In M. J. Butler, K. Schewe, A. Mashkoor, and M. Biró, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*, volume 9675 of *Lecture Notes in Computer Science (LNCS)*, pages 167–182, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

Contributions:

- Parts of the discussion on data import and representation.
- Parts of the discussion on the usage of LET and if-then-else and alternatives. The integration of these concepts into PROB was done by Dominik Hansen.
- Discussion about the use of `DEFINITIONS`.
- Discussion of the use of external functions.

The work on the validation of railway topologies discussed in this article was conducted independently by Dominik Hansen.

B
Towards B as a High-Level Constraint Modelling Language

The Alloy models used in this chapter were implemented by Daniel Plagge.

B.1 Alloy *n-Queens* Model

```
abstract sig Queens {
  row : one Int,
  col: one Int,
} {
 row ≥ 0 and row < 8
 and col ≥ 0 and col < 8
}

sig BQueens extends Queens {} {}
sig WQueens extends Queens {} {}

pred nothreat(q1,q2 : Queens) {
  q1.row ≠ q2.row
  and q1.col ≠ q2.col
  and plus[ int[q1.row] , int[q1.col]] ≠ plus[ int[q2.col] ,
      int[q2.row]]
      and plus [int[q1.row] , int[q2.col]] ≠ plus[ int[q1.col] ,
         int[q2.row]]
}

pred nothreats { all q1: BQueens, q2: WQueens • nothreat[q1, q2] }

pred alldiffB { all q1: BQueens, q2: BQueens • q1=q2 or q1.row ≠
    q2.row or q1.col ≠ q2.col }
```

```
pred alldiffW { all q1: WQueens, q2: WQueens • q1=q2 or q1.row ≠
    q2.row or q1.col ≠ q2.col }

pred equalnum {
    #(WQueens) = #(BQueens)
}

pred valid {
  nothreats and equalnum and alldiffB and alldiffW
}

fact {
  #Queens = 20
}

run valid for 20 Queens, 7 int
```

B.2 Alloy *Peaceable Armies of Queens* Model

```
sig Queens {
  row : one Int,
  col: one Int
} {
 row ≥ 0 and row < #Queens
 and col ≥ 0 and col < #Queens
}
pred nothreat(q1,q2 : Queens) {
  q1.row ≠ q2.row
  and q1.col ≠ q2.col
  and plus[int[q1.row], minus[int[q2.col], int[q1.col]]] ≠
      int[q2.row]
  and minus[int[q1.row], plus[int[q2.col], int[q1.col]]] ≠
      int[q2.row]
}
pred valid { all q1,q2 : Queens •
    q1 ≠ q2  ⟹  nothreat[q1, q2]
 }
fact card {#Queens = 8}
run valid for 8 Queens, 5 int
```

C B Machine to Validate Module Combinations for Different Courses

```
MACHINE ModuleCombinations
SEES Levels, data
CONSTANTS
  level_available_modules,
  level_mandatory_modules,
  module_combinations
PROPERTIES
  level_available_modules = {idx, mms|#(course, tt, ll).(
    (course, tt) : course_levels & ll : dom(tt)
    &
    idx = tt(ll)'idx
    &
    mms = level_modules[{x| #(y).(y : ran(subtree(tt, ll)) & x =
        y'idx)}])}
  & level_mandatory_modules = λ(idx).(idx:
      dom(level_available_modules) | {m|
                              m : level_available_modules(idx)
                              &
                              modules(m)'mandatory = TRUE})
  & module_combinations = λ(cc).(cc : course_names | dom({rv, foo|
      CHOOSE_MODULES(cc, rv, foo)}))
DEFINITIONS
  "preferences.def";
  "LibraryIO.def";

  BV(DD) == (DD → BOOL);

  CHOOSE_MODULES(course, return_value, choice) == (
    course : course_names
```

```
& courses(course)'credit_points = -1
& LET
  mm, trees
BE
  mm = course_modules(course)
  &
  trees = course_levels[{course}]
IN
  /* we are looking for a subset of all modules for the given
      course */
  /* the selection of modules must satisfy the tree-conditions
      for all trees in that course */
  /* return value is the set of all modules for the given course,
      according to the choice function */
  return_value <: mm
  & return_value = UNION(ttu).(ttu:trees| choice(ttu)~[{TRUE}])
  & choice : trees → BV(mm)
  & !(tt).(tt : trees ⇒
    LET
      lam, lmm
    BE
      lam = level_available_modules(top(tt)'idx)
      & lmm = level_mandatory_modules(top(tt)'idx)
    IN
      /* Mandatory modules have to be chosen */
      !(dd).(dd : lmm ⇒ choice(tt)(dd)=TRUE)
      & /* Modules outside the available ones are never chosen */
      !(dd).(dd : mm \ lam ⇒ choice(tt)(dd)=FALSE)
      & /* the number of modules chosen at the root level must be
          in the limits of from .. to */
      card({dd|dd:lam & dd↦TRUE:choice(tt)}) : top(tt)'from ..
          top(tt)'to
      & !(level_info).(level_info : ran(tt) ⇒
        LET
          llmm, ll_mandatory
        BE
          llmm = level_available_modules(level_info'idx)
          &
          ll_mandatory = level_mandatory_modules(level_info'idx)
        IN
          /* Mandatory modules have to be chosen */
```

```
              !(dd).(dd : ll_mandatory ⇒ choice(tt)(dd)=TRUE)
              &
              /* the number of modules chosen at each level must be in
                  the limits of from .. to */
              card({y| y: mm & y↦TRUE : choice(tt) & y : llmm}) :
                  level_info'from .. level_info'to
            END
          )
        END
      )
    END
  );
ASSERTIONS
  !(x,y).(x : dom(course_module_combinations) & y :
      course_module_combinations(x)
              ⇒ y : module_combinations(x))
END
```

D Evaluation

D.1 Data Sets

Table D.1: *Complexity index*, feasibility, number of mandatory, elective and combinations of elective modules for each course in the *Arts & Humanities* and *Business Administration & Economics* data sets used for evaluation.

Course	Feasible?	CI	Mandatory Modules	Elective Modules	Elective Module Combinations
		Business Administration & Economics			
bwl_bachelor	✓	2067530678	12	31	4495
bwl_master	✓	180153	5	25	300
vwl_bachelor	✓	475079582	11	31	4495
vwl_master	✓	217203	4	26	325
wichem_bachelor	✓	32265043	8	16	560
wichem_master	✓	8352	2	10	45
		Arts & Humanities			
ang_ger	✓	1612431360	1	0	0
ang_ges	✓	100776960	1	0	0
ang_inf	✓	201553920	1	0	0
ang_jap	✓	12538266255360	1	0	0
ang_jid	✓	6449725440	1	0	0
ang_jud	✓	403107840	1	0	0
ang_kom	✓	3627970560	1	0	0
ang_lin	✗	1612431360	1	0	0
ang_pol	✗	3627970560	1	0	0
ang_rom	✓	309586821120	1	0	0
ang_soz	✓	3627970560	1	0	0

Table D.1: *Complexity index*, feasibility, number of mandatory, elective and combinations of elective modules for each course in the *Arts & Humanities* and *Business Administration & Economics* data sets used for evaluation.

Course	Feasible?	CI	Mandatory Modules	Elective Modules	Elective Module Combinations
ger_ang	✓	4976640	1	0	0
ger_ges	✓	16	1	0	0
ger_inf	✓	32	1	0	0
ger_jap	✓	1990656	1	0	0
ger_jid	✓	1024	1	0	0
ger_jud	✓	64	1	0	0
ger_kom	✓	576	1	0	0
ger_lin	✓	256	1	0	0
ger_pol	✓	576	1	0	0
ger_rom	✓	49152	1	0	0
ger_soz	✓	576	1	0	0
ges_ang	✓	311040	1	0	0
ges_ger	✓	16	1	0	0
ges_inf	✓	2	1	0	0
ges_jap	✓	124416	1	0	0
ges_jid	✓	64	1	0	0
ges_jud	✓	4	1	0	0
ges_kom	✗	36	1	0	0
ges_lin	✗	16	1	0	0
ges_pol	✗	36	1	0	0
ges_rom	✓	3072	1	0	0
ges_soz	✗	36	1	0	0
jap_ang	✓	38698352640	1	0	0
jap_ger	✓	1990656	1	0	0
jap_ges	✓	124416	1	0	0
jap_inf	✓	248832	1	0	0
jap_jid	✗	7962624	1	0	0
jap_jud	✗	497664	1	0	0
jap_kom	✗	4478976	1	0	0
jap_lin	✗	1990656	1	0	0
jap_pol	✓	4478976	1	0	0
jap_rom	✗	382205952	1	0	0
jap_soz	✓	4478976	1	0	0

Table D.1: *Complexity index*, feasibility, number of mandatory, elective and combinations of elective modules for each course in the *Arts & Humanities* and *Business Administration & Economics* data sets used for evaluation.

Course	Feasible?	CI	Mandatory Modules	Elective Modules	Elective Module Combinations
jud_ang	✓	1244160	1	0	0
jud_ger	✓	64	1	0	0
jud_ges	✓	4	1	0	0
jud_inf	✓	8	1	0	0
jud_jap	✗	497664	1	0	0
jud_jid	✓	256	1	0	0
jud_kom	✓	144	1	0	0
jud_lin	✗	64	1	0	0
jud_pol	✓	144	1	0	0
jud_rom	✓	12288	1	0	0
jud_soz	✓	144	1	0	0
rom_ang	✓	955514880	1	0	0
rom_ger	✓	49152	1	0	0
rom_ges	✓	3072	1	0	0
rom_inf	✗	6144	1	0	0
rom_jap	✗	382205952	1	0	0
rom_jid	✗	196608	1	0	0
rom_jud	✓	12288	1	0	0
rom_kom	✓	110592	1	0	0
rom_lin	✗	49152	1	0	0
rom_pol	✓	110592	1	0	0
rom_soz	✓	110592	1	0	0
sowi	✓	46656	1	0	0

D.2 Prolog

Table D.2: SICStus brute-force and clp(FD) solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Brute-Force		clp(FD)		CI
		Runtime[‡]	SD	Runtime[‡]	SD	
Business Administration & Economics						
bwl_bachelor	✓	`timeout`[†]	-	1237[b]	±68.00	2067530678
bwl_master	✓	1242[b]	±55.94	2	±4.22	180153
vwl_bachelor	✓	`timeout`[†]	-	4	±5.16	475079582
vwl_master	✓	128	±4.22	0[a]	±0.00	217203
wichem_bachelor	✓	`timeout`[†]	-	41	±3.16	32265043
wichem_master	✓	1[a]	±3.16	2	±4.22	8352
Arts & Humanities						
ang_ger	✓	33113	±331.70	0[a]	±0.00	1612431360
ang_ges	✓	509267	±5046.97	3	±4.83	100776960
ang_inf	✓	`timeout`[†]	-	3	±4.83	201553920
ang_jap	✓	`timeout`[†]	-	104041[b]	±505.84	12538266255360
ang_jid	✓	`timeout`[†]	-	4	±5.16	6449725440
ang_jud	✓	`timeout`[†]	-	0[a]	±0.00	403107840
ang_kom	✓	`timeout`[†]	-	0[a]	±0.00	3627970560
ang_lin	✗	`timeout`[†]	-	1	±3.16	1612431360
ang_pol	✗	`timeout`[†]	-	2	±4.22	3627970560

‡ Runtime in milliseconds.
† Each timeout represents a runtime of 30 minutes.
a Minimum runtime.
b Maximum runtime.

Table D.2: SICStus brute-force and clp(FD) solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Brute-Force Runtime[‡]	Brute-Force SD	clp(FD) Runtime[‡]	clp(FD) SD	CI
ang_rom	✓	`timeout`[†]	-	10	±4.71	309586821120
ang_soz	✓	`timeout`[†]	-	4	±5.16	3627970560
ger_ang	✓	4	±5.16	2	±4.22	4976640
ger_ges	✓	1	±3.16	3	±4.83	16
ger_inf	✓	1	±3.16	1	±3.16	32
ger_jap	✓	22	±4.22	495	±21.21	1990656
ger_jid	✓	1	±3.16	2	±4.22	1024
ger_jud	✓	0[a]	±0.00	1	±3.16	64
ger_kom	✓	0[a]	±0.00	2	±4.22	576
ger_lin	✓	0[a]	±0.00	0[a]	±0.00	256
ger_pol	✓	0[a]	±0.00	0[a]	±0.00	576
ger_rom	✓	2	±4.22	1	±3.16	49152
ger_soz	✓	0[a]	±0.00	20	±0.00	576
ges_ang	✓	1296	±15.78	12	±4.22	311040
ges_ger	✓	0[a]	±0.00	1	±3.16	16
ges_inf	✓	2	±4.22	0[a]	±0.00	2
ges_jap	✓	1	±3.16	2	±4.22	124416
ges_jid	✓	2	±4.22	0[a]	±0.00	64
ges_jud	✓	1	±3.16	2	±4.22	4
ges_kom	✗	3	±4.83	1	±3.16	36
ges_lin	✗	4	±5.16	0[a]	±0.00	16
ges_pol	✗	4	±5.16	1	±3.16	36

[‡] Runtime in milliseconds.
[†] Each timeout represents a runtime of 30 minutes.
[a] Minimum runtime.
[b] Maximum runtime.

Table D.2: SICStus brute-force and clp(FD) solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Brute-Force		clp(FD)		CI
		Runtime[‡]	SD	Runtime[‡]	SD	
ges_rom	✓	1	±3.16	4	±5.16	3072
ges_soz	✗	3	±4.83	0[a]	±0.00	36
jap_ang	✓	timeout[†]	-	1225	±30.64	38698352640
jap_ger	✓	21	±3.16	496	±19.55	1990656
jap_ges	✓	1	±3.16	1	±3.16	124416
jap_inf	✓	6468	±73.00	3	±4.83	248832
jap_jid	✗	810075	±3793.82	0[a]	±0.00	7962624
jap_jud	✗	40200	±324.48	709	±18.53	497664
jap_kom	✗	890961[b]	±6464.23	0[a]	±0.00	4478976
jap_lin	✗	212611	±1553.00	147	±6.75	1990656
jap_pol	✓	135	±7.07	195	±11.79	4478976
jap_rom	✓	timeout[†]	-	0[a]	±0.00	382205952
jap_soz	✓	66151	±516.08	2	±4.22	4478976
jud_ang	✓	4362	±37.06	0[a]	±0.00	1244160
jud_ger	✓	1	±3.16	0[a]	±0.00	64
jud_ges	✓	0[a]	±0.00	0[a]	±0.00	4
jud_inf	✓	0[a]	±0.00	2	±4.22	8
jud_jap	✗	35693	±304.34	0[a]	±0.00	497664
jud_jid	✓	9	±5.68	0[a]	±0.00	256
jud_kom	✓	3	±4.83	0[a]	±0.00	144
jud_lin	✗	9	±3.16	2	±4.22	64
jud_pol	✓	0[a]	±0.00	1	±3.16	144

[‡] Runtime in milliseconds.
[†] Each timeout represents a runtime of 30 minutes.
[a] Minimum runtime.
[b] Maximum runtime.

Table D.2: SICStus brute-force and clp(FD) solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Brute-Force		clp(FD)		CI
		Runtime[‡]	SD	Runtime[‡]	SD	
jud_rom	✓	1	±3.16	3	±4.83	12288
jud_soz	✓	0[a]	±0.00	2	±4.22	144
rom_ang	✓	13766	±78.34	6	±5.16	955514880
rom_ger	✓	2	±4.22	3	±4.83	49152
rom_ges	✓	2	±4.22	1	±3.16	3072
rom_inf	✗	1871	±67.73	2	±4.22	6144
rom_jap	✓	timeout[†]	-	3	±4.83	382205952
rom_jid	✗	69347	±326.43	18	±4.22	196608
rom_jud	✓	2	±4.22	4	±5.16	12288
rom_kom	✓	2	±4.22	6	±5.16	110592
rom_lin	✗	20327	±167.07	3	±4.83	49152
rom_pol	✓	36	±5.16	0[a]	±0.00	110592
rom_soz	✓	2	±4.22	2	±4.22	110592
sowi	✓	1	±3.16	0[a]	±0.00	46656

[‡] Runtime in milliseconds.
[†] Each timeout represents a runtime of 30 minutes.
[a] Minimum runtime.
[b] Maximum runtime.

D.3 SMT-LIB/Z3

Table D.3: Z3 solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Runtime in ms.	SD	CI
		Business Administration & Economics		
bwl_bachelor	✓	110.275	±0.468	2067530678
bwl_master	✓	86.898	±0.655	180153
vwl_bachelor	✓	113.974[b]	±1.015	475079582
vwl_master	✓	87.151	±3.321	217203
wichem_bachelor	✓	106.389	±4.293	32265043
wichem_master	✓	82.735[a]	±0.616	8352
		Arts & Humanities		
ang_ger	✓	21.313	±0.861	1612431360
ang_ges	✓	22.275	±0.722	100776960
ang_inf	✓	28.585	±0.727	201553920
ang_jap	✓	32.141	±0.765	12538266255360
ang_jid	✓	27.597	±0.215	6449725440
ang_jud	✓	25.903	±0.340	403107840
ang_kom	✓	26.025	±0.279	3627970560
ang_lin	✗	0.947[a]	±0.007	1612431360
ang_pol	✗	0.974	±0.071	3627970560
ang_rom	✓	29.614	±0.074	309586821120
ang_soz	✓	27.888	±0.124	3627970560
ger_ang	✓	19.741	±0.356	4976640
ger_ges	✓	16.033	±0.487	16
ger_inf	✓	23.891	±0.273	32
ger_jap	✓	25.471	±0.270	1990656
ger_jid	✓	20.092	±0.251	1024
ger_jud	✓	14.668	±0.073	64
ger_kom	✓	13.534	±0.157	576
ger_lin	✓	24.164	±0.913	256
ger_pol	✓	16.541	±0.177	576
ger_rom	✓	20.737	±0.266	49152
ger_soz	✓	15.174	±0.110	576
ges_ang	✓	33.560	±0.454	311040

[a] Minimum runtime.
[b] Maximum runtime.

Table D.3: Z3 solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Runtime in ms.	SD	CI
ges_ger	✓	12.153	±0.271	16
ges_inf	✓	26.274	±0.432	2
ges_jap	✓	24.434	±0.315	124416
ges_jid	✓	21.764	±0.765	64
ges_jud	✓	12.718	±0.204	4
ges_kom	✗	0.980	±0.123	36
ges_lin	✗	1.001	±0.076	16
ges_pol	✗	1.020	±0.146	36
ges_rom	✓	20.023	±0.248	3072
ges_soz	✗	0.954	±0.010	36
jap_ang	✓	35.044[b]	±1.176	38698352640
jap_ger	✓	26.449	±0.837	1990656
jap_ges	✓	26.833	±0.242	124416
jap_inf	✓	29.584	±83.82	248832
jap_jid	✗	0.952	±4.22	7962624
jap_jud	✗	1.141	±0.164	497664
jap_kom	✗	0.984	±0.101	4478976
jap_lin	✗	0.972	±0.063	1990656
jap_pol	✓	29.284	±0.111	4478976
jap_rom	✗	0.982	±0.116	382205952
jap_soz	✓	27.646	±0.107	4478976
jud_ang	✓	22.953	±0.289	1244160
jud_ger	✓	14.134	±0.198	64
jud_ges	✓	18.815	±0.280	4
jud_inf	✓	26.648	±0.913	8
jud_jap	✗	1.039	±0.170	497664
jud_jid	✓	19.459	±0.377	256
jud_kom	✓	15.716	±0.597	144
jud_lin	✗	1.009	±0.125	64
jud_pol	✓	17.119	±0.444	144
jud_rom	✓	22.418	±0.251	12288
jud_soz	✓	16.452	±0.445	144
rom_ang	✓	30.541	±0.121	955514880
rom_ger	✓	22.233	±0.301	49152
rom_ges	✓	21.251	±0.316	3072

[a] Minimum runtime.
[b] Maximum runtime.

Table D.3: Z3 solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Runtime in ms.	SD	CI
rom_inf	✗	0.989	±0.077	6144
rom_jap	✗	1.012	±0.106	382205952
rom_jid	✗	0.948	±0.006	196608
rom_jud	✓	22.583	±0.324	12288
rom_kom	✓	25.787	±0.869	110592
rom_lin	✗	0.986	±0.100	49152
rom_pol	✓	22.025	±0.807	110592
rom_soz	✓	20.170	±0.362	110592
sowi	✓	15.124	±0.419	46656

[a] Minimum runtime.
[b] Maximum runtime.

D.4 Python

Table D.4: Python 2, Python 3 and PyPy solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Python 2		Python 3		PyPy		CI
		Runtime[‡]	SD	Runtime[‡]	SD	Runtime[‡]	SD	
				Business Administration & Economics				
bwl_bachelor	timeout[†]	-	-	-	-	-	-	2067530678
bwl_master	✓	0.83	±0.1	2523.54	±6856.63	10.61	±0.88	180153
vwl_bachelor	✓	2550.88	±65.42	timeout[†]	-	4079.46[b]	±176.96	475079582
vwl_master	✓	0.89	±0.017	3608.24[b]	±11006.57	1.30 [a]	±0.11	217203
wichem_bachelor	✓	102599.04[b]	±1435.42	timeout[†]	-	2.66	±0.22	32265043
wichem_master	✓	0.65[a]	±0.09	4.60[a]	±6.71	27.04	±1.54	8352
				Arts & Humanities				
ang_ger	✓	18269.50	±150.57	77729.28	±100349.57	727.39	±48.53	1612431360
ang_ges	✓	270981.06	±4494.46	1498607.53[b]	±1784865.99	389613.35	±3139.16	100776960
ang_inf	timeout[†]	-	-	-	-	-	-	201553920
ang_jap	timeout[†]	-	-	-	-	-	-	12538266255360
ang_jid	timeout[†]	-	-	-	-	-	-	6449725440
ang_jud	✓	2929.73	±60.31	9499.34	±12106.49	998072.90[b]	±8646.66	403107840
ang_kom	timeout[†]	-	-	-	-	-	-	3627970560
ang_lin	timeout[†]	-	-	-	-	-	-	1612431360
ang_pol	timeout[†]	-	-	-	-	-	-	3627970560
ang_rom	timeout[†]	-	-	-	-	-	-	309586821120
ang_soz	timeout[†]	-	-	-	-	-	-	3627970560

[‡] Runtime in milliseconds.
[†] Each timeout represents a runtime of 30 minutes.
[a] Minimum runtime.
[b] Maximum runtime.

Table D.4: Python 2, Python 3 and PyPy solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Python 2		Python 3		PyPy		CI
		Runtime[‡]	SD	Runtime[‡]	SD	Runtime[‡]	SD	
ger_ang	✓	1.05	±0.01	1.58	±0.27	0.54	±0.00	4976640
ger_ges	✓	0.25	±0.00	0.27	±0.01	0.47	±0.03	16
ger_inf	✓	0.64	±0.01	0.69	±0.03	0.48	±0.00	32
ger_jap	✓	7.81	±0.73	4.60	±1.90	298.92	±24.80	1990656
ger_jid	✓	1.22	±0.09	2.90	±3.16	0.64	±0.13	1024
ger_jud	✓	0.27	±0.01	0.30	±0.03	0.50	±0.05	64
ger_kom	✓	0.33	±0.00	0.37	±0.02	0.43	±0.00	576
ger_lin	✓	0.58	±0.00	0.68	±0.07	2.08	±0.51	256
ger_pol	✓	0.35	±0.06	0.41	±0.10	0.43	±0.01	576
ger_rom	✓	1.09	±0.01	1.62	±0.59	0.67	±0.01	49152
ger_soz	✓	0.34	±0.01	0.40	±0.07	0.41	±0.01	576
ges_ang	✓	1178.02	±23.85	1131.99	±1822.49	65.82	±9.97	311040
ges_ger	✓	0.25	±0.00	0.28	±0.02	0.36	±0.00	16
ges_inf	✓	0.54	±0.01	0.63	±0.06	1.14	±0.18	2
ges_jap	✓	1.60	±0.02	1.58	±0.34	1.50	±0.02	124416
ges_jid	✓	0.79	±0.01	0.71	±0.22	1.21	±0.04	64
ges_jud	✓	0.20[a]	±0.00	0.23[a]	±0.01	0.34[a]	±0.01	4
ges_kom	✗	1.48	±0.14	1.55	±0.21	0.61	±0.01	36
ges_lin	✗	0.74	±0.00	0.85	±0.05	1.28	±0.26	16
ges_pol	✗	1.54	±0.19	1.53	±0.11	0.55	±0.01	36
ges_rom	✓	0.96	±0.02	1.03	±0.04	0.69	±0.01	3072
ges_soz	✗	1.63	±0.15	1.59	±0.14	0.60	±0.00	36
jap_ang	timeout[†]	-	-	-	-	-	-	38698352640

‡ Runtime in milliseconds.
† Each timeout represents a runtime of 30 minutes.
a Minimum runtime.
b Maximum runtime.

Table D.4: Python 2, Python 3 and PyPy solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Python 2		Python 3		PyPy		CI
		Runtime[‡]	SD	Runtime[‡]	SD	Runtime[‡]	SD	
jap_ger	✓	7.94	±0.65	4.59	±1.96	348.43	±21.82	1990656
jap_ges	✓	1.78	±0.17	1.50	±0.26	0.65	±0.13	124416
jap_inf	✓	129.05	±2.35	37.00	±33.92	6.07	±0.96	248832
jap_jid	✗	237109.51	±1937.08	278604.28	±44212.45	34371.42	±481.77	7962624
jap_jud	✗	39207.51	±682.05	41327.29	±7440.51	4881.58	±245.77	497664
jap_kom	✗	337939.57[b]	±5700.33	351540.90	±35868.44	38805.63	±578.21	4478976
jap_lin	✗	73694.18	±476.73	79774.47	±10584.42	10291.55	±180.66	1990656
jap_pol	✓	126165.56	±1667.92	203076.17	±73462.78	5769.71	±217.50	4478976
jap_rom	timeout[†]	-	-	-	-	-	-	382205952
jap_soz	✓	30.54	±1.59	20.11	±11.36	2.19	±0.30	4478976
jud_ang	✓	1.25	±0.17	1.32	±0.05	3343.34	±64.17	1244160
jud_ger	✓	0.43	±0.06	0.43	±0.04	0.44	±0.00	64
jud_ges	✓	0.35	±0.04	0.38	±0.06	0.43	±0.12	4
jud_inf	✓	1.03	±0.14	1.02	±0.11	0.78	±0.00	8
jud_jap	✗	44418.31	±671.66	39957.74	±4275.75	4546.57	±185.92	497664
jud_jid	✓	1.01	±0.01	0.90	±0.21	0.63	±0.00	256
jud_kom	✓	2.40	±0.19	3.35	±0.69	1.09	±0.16	144
jud_lin	✗	2.16	±0.01	2.40	±0.45	1.96	±0.11	64
jud_pol	✓	0.47	±0.00	0.67	±0.14	0.56	±0.04	144
jud_rom	✓	1.35	±0.16	1.31	±0.03	0.72	±0.01	12288
jud_soz	✓	0.52	±0.06	0.67	±0.17	0.53	±0.01	144
rom_ang	✓	72734.07	±480.07	timeout[†]	-	229.60	±1.83	955514880
rom_ger	✓	14.36	±0.11	1.65	±0.77	0.74	±0.01	49152

[‡] Runtime in milliseconds.
[†] Each timeout represents a runtime of 30 minutes.
[a] Minimum runtime.
[b] Maximum runtime.

Table D.4: Python 2, Python 3 and PyPy solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Python 2		Python 3		PyPy		CI
		Runtime[‡]	SD	Runtime[‡]	SD	Runtime[‡]	SD	
rom_ges	✓	0.93	±0.01	1.02	±0.02	0.72	±0.10	3072
rom_inf	✗	204.01	±1.56	193.23	±32.41	35.27	±0.54	6144
rom_jap	timeout[†]	-	-	-	-	-	-	382205952
rom_jid	✗	4857.86	±84.84	5419.55	±974.26	882.89	±13.15	196608
rom_jud	✓	0.97	±0.03	1.12	±0.10	0.81	±0.15	12288
rom_kom	✓	1.86	±0.21	2.11	±0.65	1.82	±0.15	110592
rom_lin	✗	1062.37	±16.53	1528.30	±238.41	202.36	±2.65	49152
rom_pol	✓	1.13	±0.06	2.11	±0.78	4.88	±0.04	110592
rom_soz	✓	1.31	±0.17	2.11	±0.85	0.67	±0.04	110592
sowi	✓	81.90	±3.82	4.83	±5.13	5.52	±0.08	46656

[‡] Runtime in milliseconds.
[†] Each timeout represents a runtime of 30 minutes.
[a] Minimum runtime.
[b] Maximum runtime.

D.5 ProB

Results

Table D.5: ProB solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets.

Course	Feasible	Runtime in ms.	SD	CI
Business Administration & Economics				
bwl_bachelor	✓	387.14[b]	±28.12	2067530678
bwl_master	✓	124.29	±7.87	180153
vwl_bachelor	✓	320.00	±5.77	475079582
vwl_master	✓	101.43	±3.78	217203
wichem_bachelor	✓	238.57	±6.90	32265043
wichem_master	✓	97.14[a]	±7.56	8352
Arts & Humanities				
ang_ger	✓	71	±5.68	1612431360
ang_ges	✓	64	±5.16	100776960
ang_inf	✓	99	±5.68	201553920
ang_jap	✓	157[b]	±13.37	12538266255360
ang_jid	✓	102	±9.19	6449725440
ang_jud	✓	62	±6.32	403107840
ang_kom	✓	78	±7.89	3627970560
ang_lin	✗	61	±3.16	1612431360
ang_pol	✗	59	±5.68	3627970560
ang_rom	✓	148	±7.89	309586821120
ang_soz	✓	79	±7.38	3627970560
ger_ang	✓	46	±6.99	4976640
ger_ges	✓	13[a]	±4.83	16
ger_inf	✓	26	±5.16	32
ger_jap	✓	64	±6.99	1990656
ger_jid	✓	20	±0.00	1024
ger_jud	✓	15	±5.27	64
ger_kom	✓	20	±0.00	576
ger_lin	✓	22	±4.22	256
ger_pol	✓	20	±0.00	576
ger_rom	✓	38	±4.22	49152

[a] Minimum runtime.
[b] Maximum runtime.

Course	Feasible	Runtime in ms.	SD	CI
ger_soz	✓	20	±0.00	576
ges_ang	✓	48	±6.32	311040
ges_ger	✓	16	±5.16	16
ges_inf	✓	21	±3.16	2
ges_jap	✓	50	±4.71	124416
ges_jid	✓	20	±4.71	64
ges_jud	✓	14	±5.16	4
ges_kom	✗	18	±4.22	36
ges_lin	✗	22	±4.22	16
ges_pol	✗	17	±4.83	36
ges_rom	✓	34	±5.16	3072
ges_soz	✗	20	±4.71	36
jap_ang	✓	119	±7.38	38698352640
jap_ger	✓	67	±8.23	1990656
jap_ges	✓	50	±4.71	124416
jap_inf	✓	96	±5.16	248832
jap_jid	✗	67	±8.23	7962624
jap_jud	✗	105	±5.27	497664
jap_kom	✗	46	±6.99	4478976
jap_lin	✗	106	±14.30	1990656
jap_pol	✓	73	±4.83	4478976
jap_rom	✗	103	±6.75	382205952
jap_soz	✓	68	±6.32	4478976
jud_ang	✓	51	±5.68	1244160
jud_ger	✓	18	±4.22	64
jud_ges	✓	13[a]	±4.83	4
jud_inf	✓	29	±3.16	8
jud_jap	✗	61	±7.38	497664
jud_jid	✓	27	±4.83	256
jud_kom	✓	33	±4.83	144
jud_lin	✗	32	±6.32	64
jud_pol	✓	31	±3.16	144
jud_rom	✓	43	±4.83	12288
jud_soz	✓	31	±5.68	144
rom_ang	✓	119	±9.94	955514880
rom_ger	✓	38	±4.22	49152
rom_ges	✓	32	±4.22	3072
rom_inf	✗	53	±6.75	6144

[a] Minimum runtime.
[b] Maximum runtime.

Course	Feasible	Runtime in ms.	SD	CI
rom_jap	✗	106	±6.99	382205952
rom_jid	✗	57	±4.83	196608
rom_jud	✓	36	±5.16	12288
rom_kom	✓	46	±5.16	110592
rom_lin	✗	52	±6.32	49152
rom_pol	✓	51	±8.76	110592
rom_soz	✓	53	±6.75	110592
sowi	✓	33	±6.75	46656

[a] Minimum runtime.
[b] Maximum runtime.

Model Version 3 – Results

For comparison, we have included a table (D.6) showing the results of using the third version of our model on the latest data set provided by the faculty of humanities.[1] The presented results are averages from ten runs on our continuous integration server. On the server we run the latest versions of our models and data on top of the latest version of ProB. We report the results for checking each course independently and for each combination of major and minor. Checks were cancelled if they did not produce any results after 60 seconds.

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
Courses Checked Individually			
`BA-IWS-H-2013`	✓	1089	±17.91
`BA-KUL-H-2013`	✗	548	±12.29
`BA-LIN-COM-H-2013`	✓	3417	±38.02
`BA-LIN-GRU-H-2013`	✓	1147	±9.48
`BA-LIN-PSY-H-2013`	✓	1510	±18.25
`BA-LIN-SPR-H-2013`	✓	5702	±35.52
`BA-SMP-H-2013`	✓	41410	±190.02
`BK-ANT-N-2013`	✓	652	±11.35
`BK-AUA-H-2013`	✓	1181	±11.97
`BK-AUA-N-2013`	✓	1418	±11.35
`BK-GER-H-2013`	✓	860	±10.54
`BK-GER-KOM-N-2013`	✓	517	±8.23
`BK-GER-LANG-N-2013`	✓	524	±8.43
`BK-GER-LIT-N-2013`	✓	524	±6.99
`BK-GER-MED-N-2013`	✓	523	±8.23
`BK-GES-H-2013`	✓	1081	±9.94
`BK-GES-N-2013`	✓	323	±4.83
`BK-INF-N-2013`	✓	704	±5.16
`BK-JID-N-2013`	✓	566	±15.05
`BK-JUED-H-2013`	✓	835	±8.49
`BK-JUED-N-2013`	✓	558	±11.35
`BK-KOM-N-2013`	✓	670	±8.16
`BK-KUN-H-2013`	✓	584	±10.74

[1]Based on tag `v3-results` (`https://github.com/plues/data/tree/v3-results`)

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
`BK-KUN-N-2013`	✓	270	±4.71
`BK-LIN-GER-N-2013`	✓	306	±5.16
`BK-LIN-N-2013`	✓	304	±5.16
`BK-MOD-H-2013`	✓	994	±11.73
`BK-MOD-N-2013`	✓	285	±5.27
`BK-MOD-PLU-H-2013`	✓	530	±4.71
`BK-PHI-H-2013`	✓	642	±10.32
`BK-PHI-N-2013`	✓	459	±8.75
`BK-POL-N-2013`	✓	709	±8.75
`BK-ROM-FRA-H-2013`	✓	868	±7.88
`BK-ROM-FRA-N-2013`	✓	787	±11.59
`BK-ROM-ITA-H-2013`	✓	800	±11.54
`BK-ROM-ITA-N-2013`	✓	774	±11.73
`BK-ROM-SPA-H-2013`	✓	785	±8.49
`BK-ROM-SPA-N-2013`	✓	761	±11.97
`BK-ROMROM-FRA-N-2013`	✓	669	±11.97
`BK-ROMROM-ITA-N-2013`	✓	640	±10.54
`BK-ROMROM-SPA-N-2013`	✓	657	±9.48
`BK-RSH-N-2013`	✓	488	±9.18
`BK-SOZ-N-2013`	✓	686	±11.73
Major/Minor Combinations			
`BK-AUA-H-2013, BK-ANT-N-2013`	✓	36121	±111.10
`BK-AUA-H-2013, BK-GER-KOM-N-2013`	`timeout`	-	-
`BK-AUA-H-2013, BK-GER-LANG-N-2013`	`timeout`	-	-
`BK-AUA-H-2013, BK-GER-LIT-N-2013`	`timeout`	-	-
`BK-AUA-H-2013, BK-GER-MED-N-2013`	`timeout`	-	-
`BK-AUA-H-2013, BK-GES-N-2013`	✓	1781	±13.70
`BK-AUA-H-2013, BK-INF-N-2013`	✓	2145	±15.09
`BK-AUA-H-2013, BK-JID-N-2013`	✗	635	±10.80
`BK-AUA-H-2013, BK-JUED-N-2013`	✓	1746	±13.49
`BK-AUA-H-2013, BK-KOM-N-2013`	✓	1902	±18.13
`BK-AUA-H-2013, BK-KUN-N-2013`	✓	9282	±22.50
`BK-AUA-H-2013, BK-LIN-GER-N-2013`	✓	1655	±12.69
`BK-AUA-H-2013, BK-LIN-N-2013`	✓	1610	±14.14
`BK-AUA-H-2013, BK-MOD-N-2013`	✓	1407	±9.48

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
BK-AUA-H-2013, BK-PHI-N-2013	✓	2626	±15.05
BK-AUA-H-2013, BK-POL-N-2013	✓	2007	±14.94
BK-AUA-H-2013, BK-ROM-FRA-N-2013	✓	14018	±60.33
BK-AUA-H-2013, BK-ROM-ITA-N-2013	✓	13838	±59.59
BK-AUA-H-2013, BK-ROM-SPA-N-2013	✓	13871	±52.37
BK-AUA-H-2013, BK-ROMROM-FRA-N-2013	✓	1878	±12.29
BK-AUA-H-2013, BK-ROMROM-ITA-N-2013	✓	1849	±14.49
BK-AUA-H-2013, BK-ROMROM-SPA-N-2013	✓	1898	±17.51
BK-AUA-H-2013, BK-RSH-N-2013	timeout	-	-
BK-AUA-H-2013, BK-SOZ-N-2013	✗	1990	±12.47
BK-GER-H-2013, BK-ANT-N-2013	✓	1698	±14.75
BK-GER-H-2013, BK-AUA-N-2013	✓	3371	±28.46
BK-GER-H-2013, BK-GER-KOM-N-2013	✓	964	±19.55
BK-GER-H-2013, BK-GER-LANG-N-2013	✓	953	±9.48
BK-GER-H-2013, BK-GER-LIT-N-2013	✓	956	±15.05
BK-GER-H-2013, BK-GER-MED-N-2013	✓	951	±14.49
BK-GER-H-2013, BK-GES-N-2013	✓	1175	±12.69
BK-GER-H-2013, BK-INF-N-2013	✓	1715	±18.40
BK-GER-H-2013, BK-JID-N-2013	✓	1308	±13.16
BK-GER-H-2013, BK-JUED-N-2013	✓	1286	±12.64
BK-GER-H-2013, BK-KOM-N-2013	✓	1491	±15.95
BK-GER-H-2013, BK-KUN-N-2013	✓	1074	±12.64
BK-GER-H-2013, BK-LIN-GER-N-2013	✓	1102	±16.19
BK-GER-H-2013, BK-LIN-N-2013	✓	1092	±9.18
BK-GER-H-2013, BK-MOD-N-2013	✓	1058	±13.98
BK-GER-H-2013, BK-PHI-N-2013	✓	1098	±14.75
BK-GER-H-2013, BK-POL-N-2013	✓	1587	±17.66
BK-GER-H-2013, BK-ROM-FRA-N-2013	✓	2950	±22.60
BK-GER-H-2013, BK-ROM-ITA-N-2013	✓	2979	±21.31
BK-GER-H-2013, BK-ROM-SPA-N-2013	timeout	-	-
BK-GER-H-2013, BK-ROMROM-FRA-N-2013	✓	1509	±11.00
BK-GER-H-2013, BK-ROMROM-ITA-N-2013	✓	1442	±13.16
BK-GER-H-2013, BK-ROMROM-SPA-N-2013	✓	28786	±90.82
BK-GER-H-2013, BK-RSH-N-2013	✓	1715	±15.09
BK-GER-H-2013, BK-SOZ-N-2013	✓	1517	±16.36
BK-GES-H-2013, BK-ANT-N-2013	✓	1893	±17.02

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
BK-GES-H-2013, BK-AUA-N-2013	✓	4120	±30.91
BK-GES-H-2013, BK-GER-KOM-N-2013	✓	1627	±12.51
BK-GES-H-2013, BK-GER-LANG-N-2013	✓	1625	±11.78
BK-GES-H-2013, BK-GER-LIT-N-2013	✓	1658	±14.75
BK-GES-H-2013, BK-GER-MED-N-2013	✓	1626	±15.05
BK-GES-H-2013, BK-INF-N-2013	✓	2293	±22.13
BK-GES-H-2013, BK-JID-N-2013	✓	1621	±16.63
BK-GES-H-2013, BK-JUED-N-2013	✓	1629	±17.28
BK-GES-H-2013, BK-KOM-N-2013	✓	2053	±18.88
BK-GES-H-2013, BK-KUN-N-2013	✓	1437	±14.18
BK-GES-H-2013, BK-LIN-GER-N-2013	✓	1367	±17.66
BK-GES-H-2013, BK-LIN-N-2013	✓	1378	±16.86
BK-GES-H-2013, BK-MOD-N-2013	✓	1337	±14.94
BK-GES-H-2013, BK-PHI-N-2013	✓	1405	±18.40
BK-GES-H-2013, BK-POL-N-2013	✗	732	±7.88
BK-GES-H-2013, BK-ROM-FRA-N-2013	✓	2337	±18.88
BK-GES-H-2013, BK-ROM-ITA-N-2013	✗	674	±12.64
BK-GES-H-2013, BK-ROM-SPA-N-2013	✓	2206	±21.18
BK-GES-H-2013, BK-ROMROM-FRA-N-2013	✓	1999	±22.33
BK-GES-H-2013, BK-ROMROM-ITA-N-2013	✗	663	±8.23
BK-GES-H-2013, BK-ROMROM-SPA-N-2013	✓	1931	±20.24
BK-GES-H-2013, BK-RSH-N-2013	✗	2568	±17.51
BK-GES-H-2013, BK-SOZ-N-2013	✗	718	±7.88
BK-JUED-H-2013, BK-ANT-N-2013	✓	1457	±15.67
BK-JUED-H-2013, BK-AUA-N-2013	✓	3212	±27.80
BK-JUED-H-2013, BK-GER-KOM-N-2013	✓	1229	±11.97
BK-JUED-H-2013, BK-GER-LANG-N-2013	✓	1248	±11.35
BK-JUED-H-2013, BK-GER-LIT-N-2013	✓	1276	±15.77
BK-JUED-H-2013, BK-GER-MED-N-2013	✓	1234	±9.66
BK-JUED-H-2013, BK-GES-N-2013	✓	1146	±11.73
BK-JUED-H-2013, BK-INF-N-2013	timeout	-	-
BK-JUED-H-2013, BK-JID-N-2013	✓	1367	±14.94
BK-JUED-H-2013, BK-KOM-N-2013	✓	1443	±16.36
BK-JUED-H-2013, BK-KUN-N-2013	✓	1052	±16.19
BK-JUED-H-2013, BK-LIN-GER-N-2013	✓	1095	±14.33
BK-JUED-H-2013, BK-LIN-N-2013	✓	1075	±11.78

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
BK-JUED-H-2013, BK-MOD-N-2013	✓	1042	±10.32
BK-JUED-H-2013, BK-PHI-N-2013	✓	1092	±11.35
BK-JUED-H-2013, BK-POL-N-2013	✓	1537	±16.36
BK-JUED-H-2013, BK-ROM-FRA-N-2013	✓	3775	±31.35
BK-JUED-H-2013, BK-ROM-ITA-N-2013	✓	3748	±25.29
BK-JUED-H-2013, BK-ROM-SPA-N-2013	✓	2491	±22.33
BK-JUED-H-2013, BK-ROMROM-FRA-N-2013	✓	1434	±16.46
BK-JUED-H-2013, BK-ROMROM-ITA-N-2013	✓	1404	±14.29
BK-JUED-H-2013, BK-ROMROM-SPA-N-2013	✓	1441	±16.63
BK-JUED-H-2013, BK-RSH-N-2013	timeout	-	-
BK-JUED-H-2013, BK-SOZ-N-2013	✓	1531	±11.00
BK-KUN-H-2013, BK-ANT-N-2013	timeout	-	-
BK-KUN-H-2013, BK-AUA-N-2013	timeout	-	-
BK-KUN-H-2013, BK-GER-KOM-N-2013	✓	977	±11.59
BK-KUN-H-2013, BK-GER-LANG-N-2013	✓	906	±9.66
BK-KUN-H-2013, BK-GER-LIT-N-2013	✓	919	±11.00
BK-KUN-H-2013, BK-GER-MED-N-2013	✓	895	±10.80
BK-KUN-H-2013, BK-GES-N-2013	timeout	-	-
BK-KUN-H-2013, BK-INF-N-2013	✓	1233	±10.59
BK-KUN-H-2013, BK-JID-N-2013	✓	939	±9.94
BK-KUN-H-2013, BK-JUED-N-2013	✓	904	±14.29
BK-KUN-H-2013, BK-KOM-N-2013	✓	1117	±15.67
BK-KUN-H-2013, BK-LIN-GER-N-2013	✓	760	±14.14
BK-KUN-H-2013, BK-LIN-N-2013	✓	759	±9.94
BK-KUN-H-2013, BK-MOD-N-2013	✓	746	±11.73
BK-KUN-H-2013, BK-PHI-N-2013	✓	766	±9.66
BK-KUN-H-2013, BK-POL-N-2013	✓	1111	±8.75
BK-KUN-H-2013, BK-ROM-FRA-N-2013	✓	2682	±19.88
BK-KUN-H-2013, BK-ROM-ITA-N-2013	✓	2634	±21.18
BK-KUN-H-2013, BK-ROM-SPA-N-2013	✓	2638	±15.49
BK-KUN-H-2013, BK-ROMROM-FRA-N-2013	✓	1035	±12.69
BK-KUN-H-2013, BK-ROMROM-ITA-N-2013	✓	1009	±11.00
BK-KUN-H-2013, BK-ROMROM-SPA-N-2013	✓	1046	±11.73
BK-KUN-H-2013, BK-RSH-N-2013	✓	4646	±24.58
BK-KUN-H-2013, BK-SOZ-N-2013	✓	1049	±11.97
BK-MOD-H-2013, BK-ANT-N-2013	✓	1718	±14.75

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
`BK-MOD-H-2013, BK-AUA-N-2013`	✓	3564	±24.58
`BK-MOD-H-2013, BK-GER-KOM-N-2013`	✓	1453	±12.51
`BK-MOD-H-2013, BK-GER-LANG-N-2013`	✓	1440	±14.90
`BK-MOD-H-2013, BK-GER-LIT-N-2013`	✓	1429	±13.70
`BK-MOD-H-2013, BK-GER-MED-N-2013`	✓	1439	±14.49
`BK-MOD-H-2013, BK-GES-N-2013`	✓	1309	±17.28
`BK-MOD-H-2013, BK-INF-N-2013`	✓	1960	±19.43
`BK-MOD-H-2013, BK-JID-N-2013`	✓	1556	±12.64
`BK-MOD-H-2013, BK-JUED-N-2013`	✓	1434	±12.64
`BK-MOD-H-2013, BK-KOM-N-2013`	✓	1766	±9.66
`BK-MOD-H-2013, BK-KUN-N-2013`	✓	1129	±11.97
`BK-MOD-H-2013, BK-LIN-GER-N-2013`	✓	1268	±12.29
`BK-MOD-H-2013, BK-LIN-N-2013`	✓	1244	±15.05
`BK-MOD-H-2013, BK-PHI-N-2013`	✓	1242	±14.75
`BK-MOD-H-2013, BK-POL-N-2013`	✓	1904	±12.64
`BK-MOD-H-2013, BK-ROM-FRA-N-2013`	✓	2008	±16.19
`BK-MOD-H-2013, BK-ROM-ITA-N-2013`	✓	1983	±20.02
`BK-MOD-H-2013, BK-ROM-SPA-N-2013`	✓	2325	±16.49
`BK-MOD-H-2013, BK-ROMROM-FRA-N-2013`	✓	1714	±18.97
`BK-MOD-H-2013, BK-ROMROM-ITA-N-2013`	✓	1687	±14.94
`BK-MOD-H-2013, BK-ROMROM-SPA-N-2013`	✓	1677	±13.37
`BK-MOD-H-2013, BK-RSH-N-2013`	✓	1988	±16.86
`BK-MOD-H-2013, BK-SOZ-N-2013`	✓	1742	±16.19
`BK-MOD-PLU-H-2013, BK-ANT-N-2013`	✓	1203	±11.59
`BK-MOD-PLU-H-2013, BK-AUA-N-2013`	`timeout`	-	-
`BK-MOD-PLU-H-2013, BK-GER-KOM-N-2013`	✓	1190	±12.47
`BK-MOD-PLU-H-2013, BK-GER-LANG-N-2013`	✓	1076	±13.49
`BK-MOD-PLU-H-2013, BK-GER-LIT-N-2013`	✓	1080	±12.47
`BK-MOD-PLU-H-2013, BK-GER-MED-N-2013`	✓	1081	±12.86
`BK-MOD-PLU-H-2013, BK-GES-N-2013`	✓	854	±5.16
`BK-MOD-PLU-H-2013, BK-INF-N-2013`	✓	1364	±12.64
`BK-MOD-PLU-H-2013, BK-JID-N-2013`	✓	1090	±10.54
`BK-MOD-PLU-H-2013, BK-JUED-N-2013`	✓	1084	±8.43
`BK-MOD-PLU-H-2013, BK-KOM-N-2013`	✓	1241	±11.97
`BK-MOD-PLU-H-2013, BK-KUN-N-2013`	✓	653	±8.23
`BK-MOD-PLU-H-2013, BK-LIN-GER-N-2013`	✓	804	±8.43

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
BK-MOD-PLU-H-2013, BK-LIN-N-2013	✓	779	±11.00
BK-MOD-PLU-H-2013, BK-MOD-N-2013	✓	580	±11.54
BK-MOD-PLU-H-2013, BK-PHI-N-2013	✓	894	±13.49
BK-MOD-PLU-H-2013, BK-POL-N-2013	✓	1332	±13.16
BK-MOD-PLU-H-2013, BK-ROM-FRA-N-2013	timeout	-	-
BK-MOD-PLU-H-2013, BK-ROM-ITA-N-2013	timeout	-	-
BK-MOD-PLU-H-2013, BK-ROM-SPA-N-2013	timeout	-	-
BK-MOD-PLU-H-2013, BK-ROMROM-FRA-N-2013	✓	1306	±11.73
BK-MOD-PLU-H-2013, BK-ROMROM-ITA-N-2013	✓	1199	±11.97
BK-MOD-PLU-H-2013, BK-ROMROM-SPA-N-2013	✓	1192	±12.29
BK-MOD-PLU-H-2013, BK-RSH-N-2013	✓	5371	±30.34
BK-MOD-PLU-H-2013, BK-SOZ-N-2013	✓	1237	±15.67
BK-PHI-H-2013, BK-ANT-N-2013	timeout	-	-
BK-PHI-H-2013, BK-AUA-N-2013	timeout	-	-
BK-PHI-H-2013, BK-GER-KOM-N-2013	✓	1130	±11.54
BK-PHI-H-2013, BK-GER-LANG-N-2013	✓	1039	±12.86
BK-PHI-H-2013, BK-GER-LIT-N-2013	✓	1042	±16.86
BK-PHI-H-2013, BK-GER-MED-N-2013	✓	1042	±17.51
BK-PHI-H-2013, BK-GES-N-2013	✓	977	±17.66
BK-PHI-H-2013, BK-INF-N-2013	✓	1361	±12.86
BK-PHI-H-2013, BK-JID-N-2013	✓	993	±12.51
BK-PHI-H-2013, BK-JUED-N-2013	timeout	-	-
BK-PHI-H-2013, BK-KOM-N-2013	✓	1237	±18.28
BK-PHI-H-2013, BK-KUN-N-2013	✓	855	±13.54
BK-PHI-H-2013, BK-LIN-GER-N-2013	✓	865	±15.81
BK-PHI-H-2013, BK-LIN-N-2013	✓	847	±11.59
BK-PHI-H-2013, BK-MOD-N-2013	✓	821	±12.86
BK-PHI-H-2013, BK-POL-N-2013	✓	1239	±14.49
BK-PHI-H-2013, BK-ROM-FRA-N-2013	timeout	-	-
BK-PHI-H-2013, BK-ROM-ITA-N-2013	timeout	-	-
BK-PHI-H-2013, BK-ROM-SPA-N-2013	timeout	-	-
BK-PHI-H-2013, BK-ROMROM-FRA-N-2013	✓	1220	±24.49
BK-PHI-H-2013, BK-ROMROM-ITA-N-2013	✓	1100	±29.81
BK-PHI-H-2013, BK-ROMROM-SPA-N-2013	✓	1140	±8.16
BK-PHI-H-2013, BK-RSH-N-2013	✓	13903	±320.52
BK-PHI-H-2013, BK-SOZ-N-2013	✓	1180	±16.99

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
`BK-ROM-FRA-H-2013, BK-ANT-N-2013`	✓	1531	±50.43
`BK-ROM-FRA-H-2013, BK-AUA-N-2013`	✓	55069	±801.72
`BK-ROM-FRA-H-2013, BK-GER-KOM-N-2013`	✓	1342	±13.98
`BK-ROM-FRA-H-2013, BK-GER-LANG-N-2013`	✓	1335	±52.54
`BK-ROM-FRA-H-2013, BK-GER-LIT-N-2013`	✓	1334	±48.57
`BK-ROM-FRA-H-2013, BK-GER-MED-N-2013`	✓	1309	±14.49
`BK-ROM-FRA-H-2013, BK-GES-N-2013`	✓	1216	±10.74
`BK-ROM-FRA-H-2013, BK-INF-N-2013`	✓	1637	±24.06
`BK-ROM-FRA-H-2013, BK-JID-N-2013`	✓	1347	±44.48
`BK-ROM-FRA-H-2013, BK-JUED-N-2013`	✓	1345	±20.68
`BK-ROM-FRA-H-2013, BK-KOM-N-2013`	✓	1545	±20.68
`BK-ROM-FRA-H-2013, BK-KUN-N-2013`	✓	3734	±111.37
`BK-ROM-FRA-H-2013, BK-LIN-GER-N-2013`	✓	12111	±248.03
`BK-ROM-FRA-H-2013, BK-LIN-N-2013`	✓	12371	±274.20
`BK-ROM-FRA-H-2013, BK-MOD-N-2013`	✓	1155	±35.03
`BK-ROM-FRA-H-2013, BK-PHI-N-2013`	✓	18523	±507.54
`BK-ROM-FRA-H-2013, BK-POL-N-2013`	✓	1629	±137.87
`BK-ROM-FRA-H-2013, BK-ROM-ITA-N-2013`	✗	556	±9.66
`BK-ROM-FRA-H-2013, BK-ROM-SPA-N-2013`	✓	12664	±278.37
`BK-ROM-FRA-H-2013, BK-ROMROM-FRA-N-2013`	✗	386	±10.74
`BK-ROM-FRA-H-2013, BK-ROMROM-ITA-N-2013`	✗	562	±18.13
`BK-ROM-FRA-H-2013, BK-ROMROM-SPA-N-2013`	✓	11562	±270.38
`BK-ROM-FRA-H-2013, BK-RSH-N-2013`	`timeout`	-	-
`BK-ROM-FRA-H-2013, BK-SOZ-N-2013`	✓	1540	±15.63
`BK-ROM-ITA-H-2013, BK-ANT-N-2013`	✓	1401	±14.49
`BK-ROM-ITA-H-2013, BK-AUA-N-2013`	✓	2893	±18.88
`BK-ROM-ITA-H-2013, BK-GER-KOM-N-2013`	✓	1164	±17.12
`BK-ROM-ITA-H-2013, BK-GER-LANG-N-2013`	✓	1213	±94.99
`BK-ROM-ITA-H-2013, BK-GER-LIT-N-2013`	✓	1168	±15.49
`BK-ROM-ITA-H-2013, BK-GER-MED-N-2013`	✓	1183	±46.67
`BK-ROM-ITA-H-2013, BK-GES-N-2013`	`timeout`	-	-
`BK-ROM-ITA-H-2013, BK-INF-N-2013`	✓	2067	±17.66
`BK-ROM-ITA-H-2013, BK-JID-N-2013`	✓	1449	±13.70
`BK-ROM-ITA-H-2013, BK-JUED-N-2013`	✓	1188	±16.19
`BK-ROM-ITA-H-2013, BK-KOM-N-2013`	✓	1405	±14.33
`BK-ROM-ITA-H-2013, BK-KUN-N-2013`	✓	4310	±40.55

Table D.6: Results for running the third version of our models with ProB on the current *Arts & Humanities*data set.

Course(s)	Result	Runtime in ms.	SD
`BK-ROM-ITA-H-2013, BK-LIN-GER-N-2013`	✓	1011	±11.97
`BK-ROM-ITA-H-2013, BK-LIN-N-2013`	✓	1003	±23.11
`BK-ROM-ITA-H-2013, BK-MOD-N-2013`	✓	984	±14.29
`BK-ROM-ITA-H-2013, BK-PHI-N-2013`	✓	985	±9.71
`BK-ROM-ITA-H-2013, BK-POL-N-2013`	✗	596	±5.16
`BK-ROM-ITA-H-2013, BK-ROM-FRA-N-2013`	✗	542	±10.32
`BK-ROM-ITA-H-2013, BK-ROM-SPA-N-2013`	✓	15663	±65.15
`BK-ROM-ITA-H-2013, BK-ROMROM-FRA-N-2013`	✗	555	±12.69
`BK-ROM-ITA-H-2013, BK-ROMROM-ITA-N-2013`	✗	385	±10.80
`BK-ROM-ITA-H-2013, BK-ROMROM-SPA-N-2013`	✓	14515	±65.36
`BK-ROM-ITA-H-2013, BK-RSH-N-2013`	✓	26753	±870.31
`BK-ROM-ITA-H-2013, BK-SOZ-N-2013`	✓	1358	±9.18
`BK-ROM-SPA-H-2013, BK-ANT-N-2013`	✓	2670	±17.63
`BK-ROM-SPA-H-2013, BK-AUA-N-2013`	✓	2623	±16.36
`BK-ROM-SPA-H-2013, BK-GER-KOM-N-2013`	✓	59070	±194.65
`BK-ROM-SPA-H-2013, BK-GER-LANG-N-2013`	✓	1120	±26.24
`BK-ROM-SPA-H-2013, BK-GER-LIT-N-2013`	✓	1108	±11.35
`BK-ROM-SPA-H-2013, BK-GER-MED-N-2013`	✓	1105	±12.69
`BK-ROM-SPA-H-2013, BK-GES-N-2013`	✓	1038	±4.21
`BK-ROM-SPA-H-2013, BK-INF-N-2013`	✓	1441	±8.75
`BK-ROM-SPA-H-2013, BK-JID-N-2013`	✓	1362	±7.88
`BK-ROM-SPA-H-2013, BK-JUED-N-2013`	✓	1150	±15.63
`BK-ROM-SPA-H-2013, BK-KOM-N-2013`	✓	1372	±16.86
`BK-ROM-SPA-H-2013, BK-KUN-N-2013`	`timeout`	-	-
`BK-ROM-SPA-H-2013, BK-LIN-GER-N-2013`	✓	1074	±12.64
`BK-ROM-SPA-H-2013, BK-LIN-N-2013`	✓	968	±17.51
`BK-ROM-SPA-H-2013, BK-MOD-N-2013`	✓	960	±14.90
`BK-ROM-SPA-H-2013, BK-PHI-N-2013`	✓	980	±14.14
`BK-ROM-SPA-H-2013, BK-POL-N-2013`	✓	1668	±19.88
`BK-ROM-SPA-H-2013, BK-ROM-FRA-N-2013`	✓	12144	±127.20
`BK-ROM-SPA-H-2013, BK-ROM-ITA-N-2013`	✓	11952	±143.43
`BK-ROM-SPA-H-2013, BK-ROMROM-FRA-N-2013`	✓	12011	±300.97
`BK-ROM-SPA-H-2013, BK-ROMROM-ITA-N-2013`	✓	10782	±146.72
`BK-ROM-SPA-H-2013, BK-ROMROM-SPA-N-2013`	✗	377	±11.59
`BK-ROM-SPA-H-2013, BK-RSH-N-2013`	✓	20403	±549.06
`BK-ROM-SPA-H-2013, BK-SOZ-N-2013`	✓	1344	±19.55

Bibliography

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, 2nd edition, 1996.

[2] R. Abo and L. Voisin. Formal Implementation of Data Validation for Railway Safety-Related Systems with OVADO. In S. Counsell and M. Núñez, editors, *Software Engineering and Formal Methods - SEFM 2013*, volume 8368 of *Lecture Notes in Computer Science (LNCS)*, pages 221–236, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[3] J. Abrial. *The B-Book - Assigning Programs to Meanings.* Cambridge University Press, Cambridge, 1st edition, November 2005.

[4] J. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, Cambridge, 2010.

[5] R. J. A. Achá and R. Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, 218(1):71–91, 2014.

[6] K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, Cambridge, 2003.

[7] R. B. Ayed, S. C. Dutilleul, P. Bon, A. Idani, and Y. Ledru. B Formal Validation of ERTMS/ETCS Railway Operating Rules. In Y. A. Ameur and K.-D. Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science (LNCS)*, pages 124–129, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[8] F. Badeau and A. Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In H. Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science (LNCS)*, pages 334–354, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[9] F. Badeau and M. Doche-Petit. Formal Data Validation with Event-B. *CoRR*, abs/1210.7039, November 2012. In Proceedings of DS-Event-B 2012: Workshop on the experience of and advances in developing dependable systems in Event-B, in conjunction with ICFEM 2012 - Kyoto, Japan, November 13, 2012.

[10] G. J. Badros, A. Borning, and P. J. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(4):267–306, 2001.

[11] M. Banbara, T. Soh, N. Tamura, K. Inoue, and T. Schaub. Answer Set Programming as a Modeling Language for Course Timetabling. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5):783–798, 2013.

[12] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*, volume 39 of *International Series in Operations Research & Management Science*. Springer Science & Business Media, New York, 2012.

[13] M. Barnett, B. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science (LNCS)*, pages 364–387, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[14] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, Iowa City, 2015. Available at `http://www.smt-lib.org` - [Online; accessed 31-March-2017].

[15] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer Science & Business Media, London, 2008.

[16] J. Bendisposto, J. Clark, I. Dobrikov, P. Körner, S. Krings, L. Ladenberger, M. Leuschel, and D. Plagge. ProB 2.0 Tutorial. In M. Butler, S. Hallerstede, and M. Walden, editors, *Proceedings of the 4th Rodin User and Developer Workshop*, TUCS Lecture Notes, Turku, June 2013. Turku Centre for Computer Science.

[17] J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia. La validation de modèles Event-B avec le plug-in ProB pour RODIN. *Technique et Science Informatiques (TSI)*, 27(8):1065–1084, 2008.

[18] A. Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. In A. Balint, A. Belov, M. J. Heule, and M. Jarvisalo, editors, *Proceedings of SAT COMPETITION 2013: Solver and Benchmark Descriptions*, Helsinki, 2013. University of Helsinki - Department of Computer Science. Technical Report B-2013-1.

[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science (LNCS)*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[20] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science (LNCS)*, Berlin, Heidelberg, 1978. Springer Berlin Heidelberg.

[21] N. Bjørner and K. Jayaraman. Checking Cloud Contracts in Microsoft Azure. In R. Natarajan, G. Barua, and M. R. Patra, editors, *Distributed Computing and Internet Technology - 11th International Conference, ICDCIT 2015, Bhubaneswar, India, February 5-8, 2015. Proceedings*, volume 8956 of *Lecture Notes in Computer Science (LNCS)*, pages 21–32, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[22] O. Boite. Méthode B et Validation des Invariants Ferroviaires. Master's thesis, Université Denis Diderot, 2000. Mémoire de DEA de logique et fondements de l'informatique.

[23] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In I. Rogers, editor, *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS 2009, Genova, Italy, July 6, 2009*, pages 18–25, New York, 2009. ACM.

[24] R. A. Bosch. Peaceably Coexisting Armies of Queens. *Optima (Newsletter of the Mathematical Programming Society)*, (62):6–9, June 1999.

[25] H. Bride, O. Kouchnarenko, F. Peureux, and G. Voiron. Workflow Nets Verification: SMT or CLP? In M. H. ter Beek, S. Gnesi, and A. Knapp, editors, *Critical Systems: Formal Methods and Automated Verification: Joint 21st International Workshop on Formal Methods for Industrial Critical Systems and 16th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2016, Pisa, Italy, September 26-28, 2016, Proceedings*, pages 39–55. Springer International Publishing, Cham, 2016.

[26] M. Carlsson and P. Mildner. SICStus Prolog — The first 25 years. *Theory and Practice of Logic Programming (TPLP)*, 12(1-2):35–66, September 2011.

[27] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium,*

PLILP'97, Southampton, UK, September 3-5, 1997, Proceedings, volume 1292 of *Lecture Notes in Computer Science (LNCS)*, pages 191–206, Berlin, Heidelberg, 1997. Springer-Verlag.

[28] M. Carlsson, J. Widen, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. *SICStus Prolog user's manual.* Number 1. Swedish Institute of Computer Science Kista, Stockholm, December 2016. Release 4.3.5 - Available at `http://www.sics.se/sicstus` - [Online; accessed 14-March-2017].

[29] CENELEC. Railway applications-Communication, signalling and processing systems-Software for railway control and protection systems. CENELEC - EN 50128, European Committee for Electrotechnical Standardization, Brussels, June 2011.

[30] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA$^+$ proof system. In J. Giesl and R. Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science (LNCS)*, pages 142–148. Springer, 2010.

[31] ClearSy System Engineering. *Atelier B, User Manual.* Aix-en-Provence. Version 4.0. Available at `http://www.atelierb.eu/` - [Online; accessed 14-March-2017].

[32] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, 2001.

[33] A. Colmerauer and P. Roussel. The Birth of Prolog. In J. A. N. Lee and J. E. Sammet, editors, *HOPL-II The second ACM SIGPLAN conference on History of programming languages, Cambridge, Massachusetts, USA, April 20-23, 1993. Proceedings*, pages 37–52, New York, 1993. ACM.

[34] S. A. Cook. The Complexity of Theorem-proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, STOC '71, Shaker Heights, Ohio, USA, May 3-5, 1971.*, pages 151–158, New York, 1971. ACM.

[35] T. B. Cooper and J. H. Kingston. The complexity of timetable construction problems. In E. K. Burke and P. Ross, editors, *Practice and Theory of Automated Timetabling, First International Conference, Edinburgh, U.K., August 29 - September 1, 1995, Selected Papers*, volume 1153 of *Lecture Notes in Computer Science (LNCS)*, pages 283–295, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[36] D. Corne, P. Ross, and H.-L. Fang. Evolving Timetables. In L. Chambers, editor, *Practical Handbook of Genetic Algorithms: Applications*, volume 1, chapter 8, pages 219–276. CRC Press, Boca Raton, August 1995.

[37] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, January 2005.

[38] O. Czibula, H. Gu, A. Russell, and Y. Zinder. A multi-stage IP-based heuristic for class timetabling and trainer rostering. *Annals of Operations Research*, pages 1–29, December 2015.

[39] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science (LNCS)*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[40] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. SMT Solvers for Rodin. In J. Derrick, J. S. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, volume 7316 of *Lecture Notes in Computer Science (LNCS)*, pages 194–207, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[41] E. Demirović and N. Musliu. Modeling high school timetabling with bitvectors. *Annals of Operations Research*, pages 1–24, July 2016.

[42] P. Deransart, L. Cervoni, and A. Ed-Dbali. *Prolog: The Standard - Reference Manual.* Springer Berlin Heidelberg, Berlin, Heidelberg, May 1996.

[43] S. Deris, S. Omatu, and H. Ohta. Timetable planning using the constraint-based reasoning. *Computers & Operations Research*, 27(9):819–840, August 2000.

[44] L. Di Gaspero, B. McCollum, and A. Schaerf. The Second International Timetabling Competition (ITC-2007): Curriculum-based Course Timetabling (Track 3). Technical Report QUB/IEEE/Tech/ITC2007/CurriculumCTT/v1.0/1, Queens University Belfast - School of Electronics, Electrical Engineering and Computer Science, Belfast, August 2007.

[45] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science (LNCS)*, pages 81–94, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[46] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science (LNCS)*, pages 502–518. Springer, 2003.

[47] A. Fantechi, W. Fokkink, and A. Morzenti. Some Trends in Formal Methods Applications to Railway Signaling. In *Formal Methods for Industrial Critical Systems*, pages 61–84. John Wiley & Sons, Inc., 2012.

[48] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae*, 77(1-2):71–103, January 2007.

[49] J. S. Friedman. Automated timetabling for small colleges and high schools using huge integer programs. *CoRR*, abs/1612.08777, January 2017.

[50] A. M. Frisch, C. Jefferson, B. M. Hernández, and I. Miguel. The rules of constraint modelling. In L. P. Kaelbling and A. Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 109–116. Professional Book Center, 2005.

[51] T. Frühwirth. *Constraint Handling Rules.* Cambridge University Press, Cambridge, 1st edition, July 2009.

[52] H. Gallaire and J. Minker, editors. *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, 1977*, Advances in Data Base Theory, New York, 1978. Plemum Press.

[53] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In O. Nierstrasz, editor, *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, volume 707 of *Lecture Notes in Computer Science (LNCS)*, pages 406–431. Springer, 1993.

[54] Y. Ge and L. de Moura. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science (LNCS)*, pages 306–320. Springer, 2009.

[55] I. P. Gent, K. E. Petrie, and J. Puget. Symmetry in Constraint Programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, Oxford, 2006.

[56] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation Using Constraint Solving Techniques. In M. L. Soffa, M. Young, and W. Tracz, editors, *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 1998, Clearwater Beach, Florida, USA, March 2-5, 1998*, pages 53–62, New York, 1998. ACM.

[57] C. C. Gotlieb. The Construction Of Class-Teacher Timetables. In *Preprint of the proceedings of the IFIP Congress 62 / Munich, August 27*, pages 22–25, 1963.

[58] A. Gupta, M. K. Ganai, and C. Wang. SAT-Based Verification Methods and Applications in Hardware Verification. In M. Bernardo and A. Cimatti, editors, *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006, Bertinoro, Italy, May 22-27, 2006, Advanced Lectures*, pages 108–143. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[59] H. Habrias, editor. *Proceedings of the 1st Conference on the B method*, Putting into Practice Methods and Tools for Information System Design, Nantes, November 1996. IRIN Institut de recherche en informatique de Nantes.

[60] D. Hansen and M. Leuschel. Translating TLA$^+$ to B for Validation with ProB. In J. Derrick, S. Gnesi, D. Latella, and H. Treharne, editors, *Integrated Formal Methods - 9th International Conference, IFM 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, volume 7321 of *Lecture Notes in Computer Science (LNCS)*, pages 24–38, Berlin, Heidelberg, June 2012. Springer Berlin Heidelberg.

[61] D. Hansen, D. Schneider, and M. Leuschel. Using B and ProB for Data Validation Projects. In M. J. Butler, K. Schewe, A. Mashkoor, and M. Biró, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*, volume 9675 of *Lecture Notes in Computer Science (LNCS)*, pages 167–182, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[62] I. J. Hayes, C. B. Jones, and J. E. Nicholls. Understanding the Differences Between VDM and Z. *SIGSOFT Software Engineering Notes*, 19(3):75–81, July 1994.

[63] D. Herman and M. Wand. A Theory of Hygienic Macros. In S. Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4960 of *Lecture Notes in Computer Science (LNCS)*, pages 48–62, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[64] R. Hickey. The clojure programming language. In J. Brichau, editor, *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, page 1. ACM, 2008.

[65] M. Hinchey and L. Coyle. Evolving Critical Systems: A Research Agenda for Computer-Based Systems. In R. Sterritt, B. Eames, and J. Sprinkle, editors, *17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2010, Oxford, England, UK, 22-26 March 2010*, pages 430–435. IEEE Computer Society, 2010.

[66] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Upper Saddle River, 1985.

[67] G. J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004.

[68] N. Huynh. *Vérification Et Validation De Politiques De Contrôle D'accès Dans Le Domaine Médical.* PhD thesis, Université de Sherbrooke, Université de Sherbrooke/Université Paris-Est, 2016.

[69] ISO/IEC. Information technology – Programming languages – Prolog – Part 1: General core. ISO 13211-1:1995, International Organization for Standardization, Geneva, June 1995.

[70] ISO/IEC. Information technology – Z formal specification notation – Syntax, type system and semantics. ISO 13568:2002, International Organization for Standardization, Geneva, July 2002.

[71] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11:256–290, April 2002.

[72] D. Jackson. *Software Abstractions - Logic, Language, and Analysis.* MIT Press, Cambridge, 2012.

[73] E. K. Jackson, T. Levendovszky, and D. Balasubramanian. Reasoning about Metamodeling with Formal Specifications and Automatic Proofs. In J. Whittle, T. Clark, and T. Kühne, editors, *Model Driven Engineering Languages and Systems, 14th International Conference, MoDELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, volume 6981 of *Lecture Notes in Computer Science (LNCS)*, pages 653–667, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[74] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–581, 1994.

[75] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University - Software Engineering Insititue, Pittsburgh, 1990.

[76] H. A. Kautz and B. Selman. Planning as Satisfiability. In B. Neumann, editor, *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pages 359–363, Chichester, July 1992. John Wiley & Sons, Inc.

[77] J. C. Knight. Safety Critical Systems: Challenges and Directions. In W. Tracz, M. Young, and J. Magee, editors, *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 547–550. ACM, 2002.

[78] R. A. Kowalski. The Early Years Of Logic Programming. *Communications of the ACM*, 31(1):38–43, January 1988.

[79] S. Krings. *Towards Infinite-State Symbolic Model Checking for B and Event-B*. PhD thesis, Heinrich Heine University Düsseldorf, Düsseldorf, 2017. Unpublished.

[80] S. Krings, J. Bendisposto, and M. Leuschel. From Failure to Proof: The ProB Disprover for B and Event-B. In R. Calinescu and B. Rumpe, editors, *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, volume 9276 of *Lecture Notes in Computer Science (LNCS)*, pages 199–214, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[81] S. Krings and M. Leuschel. Constraint Logic Programming over Infinite Domains with an Application to Proof. In S. Schwarz and J. Voigtländer, editors, *Proceedings 29th and 30th Workshops on (Constraint) Logic Programming and 24th International Workshop on Functional and (Constraint) Logic Programming, and 24th International Workshop on Functional and (Constraint) Logic Programming, WLP 2015 / WLP 2016 / WFLP 2016, Dresden and Leipzig, Germany, 22nd September 2015 and 12-14th September 2016.*, volume 234 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 73–87, Waterloo, 2017. Open Publishing Association.

[82] G. Lach and M. E. Lübbecke. Curriculum based course timetabling: new solutions to Udine benchmark instances. *Annals of Operations Research*, 194(1):255–272, February 2012.

[83] L. Lamport. *Specifying Systems, The* TLA$^+$ *Language and Tools for Hardware and Software Engineers.* Addison-Wesley, Amsterdam, 2002.

[84] T. Lecomte, L. Burdy, and M. Leuschel. Formally Checking Large Data Sets in the Railways. *CoRR*, abs/1210.6815, November 2012. In Proceedings of DS-Event-B 2012: Workshop on the experience of and advances in developing dependable systems in Event-B, in conjunction with ICFEM 2012 - Kyoto, Japan, November 13, 2012.

[85] T. Lecomte and M. Leuschel. Formal Data Validation Tutorial. Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Presentation, June 2014.

[86] T. Lecomte and E. Mottin. Formal Data Validation in the Railways. In *Developing Safe Systems: Proceedings of the Twenty-fourth Safety-critical Systems Symposium, Brighton, UK, 2nd-4th February 2016*. CreateSpace Independent Publishing Platform, December 2015.

[87] M. Leuschel, J. Bendisposto, I. Dobrikov, S. Krings, and D. Plagge. From Animation to Data Validation: The ProB Constraint Solver 10 Years On. In J.-L. Boulanger, editor, *Formal Methods Applied to Complex Systems*, chapter 14, pages 427–446. John Wiley & Sons, Inc., Hoboken, July 2014.

[88] M. Leuschel and M. Butler. ProB: A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science (LNCS)*, pages 855–874, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[89] M. Leuschel, D. Cansell, and M. J. Butler. Validating and Animating Higher-Order Recursive Functions in B. In J. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday*, volume 5115 of *Lecture Notes in Computer Science (LNCS)*, pages 78–92. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[90] M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models with ProB. *Formal Aspects of Computing*, 23(6):683–709, 2011.

[91] M. Leuschel and M. Fontaine. Probing the Depths of CSP-M: A New FDR-Compliant Validation Tool. In S. Liu, T. S. E. Maibaum, and K. Araki, editors, *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings*, volume 5256 of *Lecture Notes in Computer Science (LNCS)*, pages 278–297. Springer, 2008.

[92] M. Leuschel, M. Samia, J. Bendisposto, and L. Luo. Easy Graphical Animation and Formula Viewing for Teaching B. In C. Attiogbé and H. Habrias, editors, *The B Method : From Research to Teaching, June 16, 2008, Nantes, France, Proceedings*, pages 17–32, June 2008.

[93] M. Leuschel and D. Schneider. Towards B as a High-Level Constraint Modelling Language - Solving The Jobs Puzzle Challenge. In Y. A. Ameur and K.-D. Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science (LNCS)*, pages 101–116, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[94] R. Lewis. A survey of metaheuristic-based techniques for University Timetabling problems. *OR Spectrum*, 30(1):167–190, 2008.

[95] B. Luteberget, C. Johansen, and M. Steffen. Rule-Based Consistency Checking of Railway Infrastructure Designs. In E. Ábrahám and M. Huisman, editors, *Integrated Formal Methods - 12th International Conference, IFM 2016, Reykjavik, Iceland, June 1-5, 2016, Proceedings*, volume 9681 of *Lecture Notes in Computer Science (LNCS)*, pages 491–507, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[96] I. Lynce and J. P. M. Silva. On Computing Minimum Unsatisfiable Cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Proceedings*, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[97] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. G. de la Banda, and M. Wallace. The Design of the Zinc Modelling Language. *Constraints*, 13(3):229–267, 2008.

[98] W. McCune. OTTER 3.3 Reference Manual. Technical Report ANL/MCS-TM-263, Argonne National Laboratory - Mathematics and Computer Science Division, 2003. Available at `http://www.cs.unm.edu/~mccune/otter/Otter33.pdf` - [Online; accessed 13-April-2017].

[99] A. Milicevic and D. Jackson. Preventing arithmetic overflows in Alloy. *Science of Computer Programming*, 94, Part 2:203–216, 2014.

[100] I. Milicevic, Aleksandar Erfrati and D. Jackson. aRby—An Embedding of Alloy in Ruby. In Y. A. Ameur and K.-D. Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science (LNCS)*, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[101] V. Montaghami and D. Rayside. Extending Alloy with Partial Instances. In J. Derrick, J. S. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, volume 7316 of *Lecture Notes in Computer Science (LNCS)*, pages 122–135, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[102] T. Müller. *Constraint-based Timetabling.* PhD thesis, Charles University, Prague, 2005.

[103] T. Müller and H. Rudová. Real-life Curriculum-based Timetabling with Elective Courses and Course Sections. *Annals of Operations Research*, 239(1):153–170, 2016.

[104] C. Newcombe. Why amazon chose TLA +. In Y. A. Ameur and K.-D. Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science (LNCS)*, pages 25–39, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[105] I. Niemelä, P. Simons, and T. Syrjänen. Smodels: A System for Answer Set Programming. *CoRR*, cs.AI/0003033, 2000. Proceedings of the 8th International Workshop on Non-Monotonic Reasoning.

[106] D. Plagge. *Supporting Validation and Verification of State-Based Formal Models.* PhD thesis, Heinrich Heine University Düsseldorf, Düsseldorf, 2016.

[107] D. Plagge and M. Leuschel. Validating Z Specifications using the ProB Animator and Model Checker. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods: 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007. Proceedings*, volume 4591 of *Lecture Notes in Computer Science (LNCS)*, pages 480–500, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[108] D. Plagge and M. Leuschel. Validating B, Z and TLA$^+$ Using ProB and Kodkod. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science (LNCS)*, pages 372–386, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[109] G. Post, L. D. Gaspero, J. H. Kingston, B. McCollum, and A. Schaerf. The Third International Timetabling Competition. *Annals of Operations Research*, 239(1):69–75, 2016.

[110] M. C. Reynolds. Lightweight Modeling of Java Virtual Machine Security Constraints. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, editors, *Abstract State Machines, Alloy, B and Z, Second International Conference, ABZ 2010,*

Orford, QC, Canada, February 22-25, 2010. Proceedings, volume 5977 of *Lecture Notes in Computer Science (LNCS)*, pages 146–159, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[111] H. Rudová and K. S. Murray. University Course Timetabling with Soft Constraints. In E. K. Burke and P. D. Causmaecker, editors, *Practice and Theory of Automated Timetabling IV, 4th International Conference, PATAT 2002, Gent, Belgium, August 21-23, 2002, Selected Revised Papers*, volume 2740 of *Lecture Notes in Computer Science (LNCS)*, pages 310–328. Springer, 2002.

[112] K. Schimmelpfeng and S. Helber. Application of a real-world university-course timetabling model solved by integer programming. *OR Spectrum*, 29(4):783–803, December 2007.

[113] D. Schneider, M. Leuschel, and T. Witt. Model-Based Problem Solving for University Timetable Validation and Improvement. In N. Bjørner and F. S. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science (LNCS)*, pages 487–495, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[114] R. Schwitter. The jobs puzzle: Taking on the challenge via controlled natural language processing. *Theory and Practice of Logic Programming (TPLP)*, 13:487–501, July 2013.

[115] T. Servat. BRAMA: A New Graphic Animation Tool for B Models. In J. Julliand and O. Kouchnarenko, editors, *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *Lecture Notes in Computer Science (LNCS)*, pages 274–276, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[116] S. C. Shapiro. The Jobs Puzzle: A Challenge for Logical Expressibility and Automated Reasoning. In *Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21-23, 2011*, Palo Alto, 2011. Association for the Advancement of Artificial Intelligence (AAAI).

[117] S. C. Shapiro and The SNePS Implementation Group. *SNePS 2.7.1 User's Manual*. Department of Computer Science and Engineering University at Buffalo, The State University of New York, Buffalo, December 2010. Available at `http://www.cse.buffalo.edu/sneps/Manuals/manual271.pdf` - [Online; accessed 13-April-2017].

[118] R. Sharpe. Formal methods start to add up again. *Computing*, 301(8):24–25, 2004.

[119] B. M. Smith, K. E. Petrie, and I. P. Gent. Models and Symmetry Breaking for 'Peaceable Armies of Queens'. In J. Régin and M. Rueher, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, volume 3011 of *Lecture Notes in Computer Science (LNCS)*, pages 271–286, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[120] J. M. Spivey. *The Z notation - A Reference Manual.* Prentice Hall International Series in Computer Science. Prentice Hall, Upper Saddle River, 1989.

[121] L. Sterling and E. Y. Shapiro. *The Art of Prolog - Advanced Programming Techniques.* MIT Press, Cambridge, 2nd edition, 1994.

[122] T. Swift and D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming (TPLP)*, 12(1-2):157–187, 2012.

[123] N. Tillmann and J. de Halleux. Pex-White Box Test Generation for .NET. In B. Beckert and R. Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science (LNCS)*, pages 134–153, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[124] E. Torlak, F. S. Chang, and D. Jackson. Finding Minimal Unsatisfiable Cores of Declarative Specifications. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science (LNCS)*, pages 326–341, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[125] E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science (LNCS)*, pages 632–647, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[126] M. Triska. The Finite Domain Constraint Solver of SWI-Prolog. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, volume 7294 of *Lecture Notes in Computer Science (LNCS)*, pages 307–316. Springer, 2012.

[127] G. van Rossum and F. L. Drake. *The Python Language Reference Manual.* Network Theory Ltd., 2011.

[128] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager. SWI-Prolog. *Theory and Practice of Logic Programming (TPLP)*, 12(1-2):67–96, 2012.

[129] L. Wos, R. Overbeck, E. Lusk, and J. Boyle. *Automated Reasoning: Introduction and Applications.* Prentice Hall, Old Tappan, 1984.

[130] V. S. H. Yeung. Declarative Configuration Applied to Course Scheduling. Master's thesis, Massachusetts Institute of Technology, Cambridge, 2006.

[131] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA$^+$ Specifications. In L. Pierre and T. Kropf, editors, *CHARME '99 - Correct Hardware Design and Verification Methods, Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999*, volume 1703 of *Lecture Notes in Computer Science (LNCS)*, pages 54–66, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[132] N. Zhou, M. Tsuru, and E. Nobuyama. A Comparison of CP, IP, and SAT Solvers through a Common Interface. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 41–48. IEEE Computer Society, 2012.

List of Figures

2.1 B Machine modelling a counter that can be incremented until it reaches its predefined maximum value. This model contains an error. 11
2.2 Screenshot of ProB's user interface 15

3.1 The solution to the Jobs puzzle, depicted graphically 29
3.2 B specification to check if two graphs are isomorphic. 38

4.1 Graphical representation of a semester week highlighting a detected scheduling conflict. Each numbered box represents an event, the numbers represent the teaching units and groups the events are associated to. The two highlighted units (65 and 67) are in conflict because they are assigned to the same curriculum while their sessions share a time slot on Tuesday at 10:30. 46
4.2 Entities and relationships in the third version of our curriculum data model annotated with sample data from Figures 4.3 and 4.9. 47
4.3 Modules and module selection rules for an exemplary course "Philosophy". Rules and modules are shown in the raw XML data format and as an equivalent graphical representation, where one of the valid module combination is highlighted. 51
4.4 Feature diagram describing the different selection rules in a curriculum. . 54
4.5 Schematic representation of the data import and transformation process: the user provided raw data is translated into an intermediate representation that serves as input for the main application. The main application generates a B representation of the data which together with the validation machine is used to check the feasibility of the data. 60
4.6 Simple tree structure with labelled edges representing the order of the child nodes. 61
4.7 Construction of a tree data structure in B. 62
4.8 B representation of level trees. 63
4.9 Validation step to check that the number of selected modules for a level-node is within the permitted bounds. 63
4.9 XML and graphical representation of entities and relationships in the curriculum data. 66
4.10 Representation of sessions, groups and units as B records. 67
4.11 Computing the set of IDs for all mandatory abstract units according to their type field. 68
4.12 Simplified conflict detection logic for a choice of units. 71

4.13 Defining the choice of group constraint using a total function (`groupChoice`). 71
4.14 Simplified version of unsatisfiable core computation in B. The first version uses a `DEFINITION` to represent if-then-else, the second versions uses ProB's if-then-else support in predicates and expressions. 74
4.15 Simplified version of computing the unsatisfiable core of a set of units using an external function. 75
4.16 Example of alternatives for the session "Basis 3/ A3a: Methodenkurs Logik". The alternatives in this example are only valid for the course "German studies/Linguistics". 77
4.17 Schematic representation of the system architecture, showing the different components and the interaction between them. 81
4.18 Interface between the application and the formal model to execute an operation using the ProB Java API. 82
4.19 Simplified server response from `GET /api/sessions` when loading all available sessions. 84
4.20 Screenshot of the timetable for the first semester including all available sessions. 84

5.1 Modelling of an engineering rule as a validation predicate. 93
5.2 Generalized architecture of ProB based data validation project 94
5.3 Calculating the conflict relation of two signals placed at the same position. 99
5.4 Using a `LET` inside a definition for scoping. 101
5.5 Query operation 103
5.6 Implementation of a sorting algorithm. 104
5.7 Recursive sort function 105
5.8 Translation of a `RULES_MACHINE` to an ordinary B machine 107

6.1 Feature diagram describing the validation and selection rules for the second version of our model. 115
6.2 Plots of the courses in each faculty ordered by their *complexity index*. The *complexity index* is presented in a logarithmic scale. 117
6.3 Artificial example of a recursive and a clp(FD) based search. 122
6.4 Prolog implementation of a search for a minimal unsatisfiable core of a list of values with regard to a given predicate. 123
6.5 Predicate to check unit compatibility with `block` annotations to create coroutines. Coroutines will wait on variables marked with `-` in the annotations. 126
6.6 Predicate that collects possible groups for all selected units and created clp(FD) variables with constrained domains in an AVL tree. 126
6.7 Checking for a potential conflict between two sessions. 127

6.8 Prolog clp(FD) and brute-force results for **feasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*. 131
6.9 Prolog clp(FD) and brute-force results for **infeasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*. 132
6.10 Prolog clp(FD) and brute-force results for courses in the *Business Administration & Economics* data set ordered by increasing *complexity index*. . 133
6.11 Example of a simple SMT-LIB model with one constant and one assertion. 136
6.12 Example of an unsatisfiable SMT-LIB model and of the unsat-core computation. 136
6.13 SMT-LIB declarations of data types used to represent time slots and semesters. 137
6.14 SMT-LIB data type and sort declarations used to represent **sessions**. . . 137
6.15 SMT-LIB function and assertions used to represent and constrain the **unit** choice. 138
6.16 SMT-LIB assertion and predicates used to check pairs of sessions for conflicts. 139
6.17 SMT-LIB of an assertion constraining the group choice for a unit to one of its groups in case it is selected. 140
6.18 Z3 results for courses in the faculty of *Business Administration & Economics* data set ordered by increasing *complexity index*. 143
6.19 Z3 results for **feasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*. 144
6.20 Z3 results for **infeasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*. 144
6.21 Example of an Alloy signature with one field and an associated fact constraining valid instances. 145
6.22 Unsatisfiable core representation in the Alloy Analyzer. 147
6.23 Alloy representation of time slots and rhythms using `enum` signatures. . . 147
6.24 Alloy representation of different **sessions**. 148
6.25 **Course** selection predicate used to set the course to be validated. 148
6.26 Curriculum signature and associated facts used to represent solutions. . . 149
6.27 Python implementation of a search for a minimal unsatisfiable core of a list of values with regard to a given predicate. 152
6.28 Python class used to represent units. 153
6.29 Python implementation of the conflict check for two units. 154
6.30 Python results for **feasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*. 159
6.31 Python results for **infeasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*. 160
6.32 Python results for *Business Administration & Economics* data set ordered by increasing *complexity index*. 160

6.33 ProB results for courses in the *Business Administration & Economics* data set ordered by increasing *complexity index*. 163
6.34 ProB results for **feasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*. 164
6.35 ProB results for **infeasible** courses in the *Arts & Humanities* data set ordered by increasing *complexity index*. 164
6.36 Relative runtimes normalized to ProB on the *Business Administration & Economics* data set. Courses displayed in ascending order by *complexity index*. 173
6.37 Relative runtimes for **feasible** courses on the *Arts & Humanities* data set normalized to ProB and shown on a logarithmic scale. 174
6.38 Relative runtimes for **infeasible** courses on the *Arts & Humanities* data set normalized to ProB and shown on a logarithmic scale. 175

List of Tables

3.1 Time in *ms.* to find a first solution to the *n-Queens* problem. 32
3.2 Implementation size in bytes for the different solutions compared 34
3.3 Runtime in milliseconds to solve the *Peaceable Queens* problem 36

4.1 All possible module combinations for the module tree in Figure 4.3. . . . 52
4.2 List of key features as supported in each version of our models. 79
4.3 Number of entities for the faculties of *Arts & Humanities* (AH) and *Business Administration & Economics* (BAE) in the second version of our model. 80
4.4 Number of entities for the faculties of *Arts & Humanities* (AH), *Business Administration & Economics* (BAE) and the *Computer Science* (CS) curricula in the third version of our model. 80

6.1 *Complexity index*, feasibility, number of mandatory, elective and combinations of elective modules for the *Business Administration & Economics* data set and for the 25 courses with the largest *complexity index* values in the *Arts & Humanities* data set. 118
6.2 SICStus brute-force and clp(FD) results for the *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set. 129
6.3 Summary of results for both Prolog based implementations. 133
6.4 Summary of results for Z3. 140
6.5 Z3 results for the *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set. 142
6.6 Summary of results for Python based implementations. 155
6.7 Python results for the *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set. . . 157
6.8 Summary of results as computed by ProB 161
6.9 ProB results for the *Business Administration & Economics* data set and 25 largest *complexity index* values in the *Arts & Humanities* data set. . . 162
6.10 Filesize of models (solver and generated data) in bytes for each faculty and language/tool. 165
6.11 Filesize in bytes of templates used to generate data models for each language/tool. 167
6.12 Summary of benchmark results. 168

D.1 *Complexity index*, feasibility, number of mandatory, elective and combinations of elective modules for each course in the *Arts & Humanities* and *Business Administration & Economics* data sets used for evaluation. . . 197
D.2 SICStus brute-force and clp(FD) solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets. 200
D.3 Z3 solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets. 204
D.4 Python 2, Python 3 and PyPy solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets. 207
D.5 ProB solver runtimes for the *Business Administration & Economics* and *Arts & Humanities* data sets. 211
D.6 Results for running the third version of our models with ProB on the current *Arts & Humanities*data set. 214