

Universidade do Minho

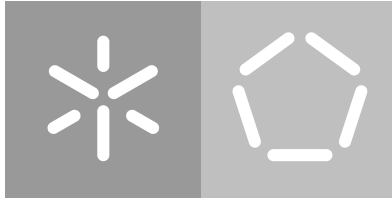
Escola de Engenharia

Departamento de Informática

Pedro Faria Durães da Silva

Towards a Quantitative Alloy

March 2021



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Pedro Faria Durães da Silva

Towards a Quantitative Alloy

Master dissertation

Integrated Master's in Informatics Engineering

Dissertation supervised by

José N. Oliveira (University of Minho)

Nuno Macedo (University of Porto)

March 2021

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

LICENSE GRANTED TO USERS OF THIS WORK:



CC BY

<https://creativecommons.org/licenses/by/4.0/>

ACKNOWLEDGEMENTS

First of all, I would like to thank Professor José Nuno for giving me the opportunity to do my MSc dissertation in such an interesting project. Further, thank you, not only for all the patience and the enormous help from early on, given my many questions at the time, making the start-off of this project as smooth as possible, but also for all the guidance throughout the project as a whole.

I also wish to thank Dr. Nuno Macedo for his crucial help in establishing a research plan that effectively allowed me to deliver the objectives of this dissertation; and for all the feedback provided, clearing up many doubts and pointing out improvements that would push this project further.

Once again, especially in this harsh circumstances faced by everyone this year, I am very thankful for all the help my supervisors were able to provide me, regarding the execution of the project itself, greatly improving the quality of my writing, and so on; without which such a degree of accomplishment would not have been possible.

I am forever indebted to my family, which were my pillars not only for all of my existence but also during this key phase of my life. Father Júlio, thank you for being there every step of the way, for being an endless source of positivity and encouragement, for all the time you have spared to assist me. Arminda, my mother, thank you for always knowing how to put a smile on my face no matter how I may feel, for always caring, for helping me keep my feet on the ground when I needed it the most. Dear sister Diana, my inspiration and role model, I strive to reach your level of excellence – thank you for always being ready to help and for always believing in me. Thank you for everything that you have done for me, for all the love and support you have always given me,

I hope to have made you proud.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

When one comes across a new problem that needs to be solved, by abstracting from its associated details in a simple and concise way through the use of formal methods, one is able to better understand the matter at hand. Alloy (Jackson, 2012), a declarative specification language based on relational logic, is an example of an effective modelling tool, allowing high-level specification of potentially very complex systems. However, along with the irrelevant information, measurable data of the system is often lost in the abstraction as well, making it not as adequate for certain situations.

The Alloy Analyzer represents the relations under analysis by Boolean matrices. By extending this type of structure to:

- numeric matrices, over \mathbb{N}_0 , one is able to work with *multirelations*, i.e. relations whose arcs are weighted; each tuple is thus associated with a natural number, which allows reasoning in a similar fashion as in *optimization problems* and *integer programming* techniques;
- left-Stochastic matrices, one is able to model faulty behaviour and other forms of quantitative information about software systems in a probabilistic way; in particular, this introduces the notion of a *probabilistic contract* in software design.

Such an increase in Alloy's capabilities strengthens its position in the area of formal methods for software design, in particular towards becoming a *quantitative formal method*.

This dissertation explores the motivation and importance behind quantitative analysis by studying and establishing theoretical foundations through *categorical* approaches to accomplish such reasoning in Alloy. This starts by reviewing the required tools to support such groundwork and proceeds to the design and implementation of such a quantitative Alloy extension.

This project aims to promote the evolution of quantitative formal methods by successfully achieving quantitative abstractions in Alloy, extending its support to these concepts and implementing them in the Alloy Analyzer.

KEYWORDS Alloy, Category Theory, Quantitative formal methods, Relational Algebra, Typed Linear Algebra.

RESUMO

Quando se depara com um novo problema que precisa de ser resolvido, ao abstrair dos seus detalhes associados de forma simples e concisa recorrendo a métodos formais, é possível compreender melhor o assunto em questão. Alloy (Jackson, 2012), uma linguagem de especificação declarativa baseada em lógica relacional, é um exemplo de uma ferramenta de modelação eficaz, possibilitando especificações de alto-nível de sistemas potencialmente bastante complexos. Contudo, em conjunto com a informação irrelevante, os dados mensuráveis são muitas vezes também perdidos na abstracção, tornando-a não tão adequada para certas situações.

O Alloy Analyzer representa as relações sujeitas a análise através de matrizes Booleanas. Ao estender este tipo de estrutura para:

- matrizes numéricas, em \mathbb{N}_0 , é possível lidar com *multirelações*, i.e., relações cujos arcos são pesados; cada tuplo é consequentemente associado a um número natural, o que proporciona uma linha de raciocínio semelhante à de técnicas de *problemas de otimização* e de *programação inteira*;
- matrizes estocásticas, permitindo a modelação de comportamento defeituoso e de outros tipos de informação quantitativa de sistemas de software probabilisticamente; em particular, é introduzida a noção de *contrato probabilístico* em design de software.

Tal aumento às capacidades do Alloy, fortalece a sua posição na área de métodos formais para design de software, em particular, a caminho de se tornar um *método formal quantitativo*.

Esta dissertação explora a motivação e a importância subjacente à análise quantitativa, a partir do estudo e consolidação dos fundamentos teóricos através de abordagens *categóricas* de forma a conseguir suportar esse tipo de raciocínio em Alloy. Inicialmente, as ferramentas imprescindíveis para assegurar tal base são analisadas, passando de seguida ao planeamento e posterior implementação de tal extensão quantitativa do Alloy.

Este projecto pretende promover a evolução dos métodos formais quantitativos através da concretização de abstracção quantitativa em Alloy, estendendo a sua base para suportar estes conceitos e assim implementá-los no Alloy Analyzer.

PALAVRAS-CHAVE Álgebra relacional, Alloy, Métodos formais quantitativos, Teoria das categorias, *Typed Linear Algebra*.

CONTENTS

1	INTRODUCTION	1
1.1	Structure of the Dissertation	6
2	BACKGROUND	8
2.1	First-order Relational Logic	8
2.2	Typed Linear Algebra	13
2.3	Boolean Satisfiability Problem	19
2.4	Satisfiability Modulo Theories	20
2.5	Probabilistic Model Checking	21
2.6	Alloy	23
2.7	Summary	28
3	STATE OF THE ART	29
3.1	Tools	29
3.1.1	Satisfiability Modulo Theory Problem Solvers	29
3.1.2	PRISM	33
3.2	Related Work	38
3.2.1	Encoding Multirelations in Alloy	38
3.2.2	Relational Algebra and SMT Solvers	44
3.3	Summary	46
4	THE PROBLEM AND ITS CHALLENGES	47
4.1	Case Studies	47
4.1.1	Bibliometrics	48
4.1.2	Football Championship	49
4.1.3	Sprinkler	54
4.2	Problem	59
4.3	Proposed Approach - Solution	60
4.3.1	System Architecture	60
4.4	Summary	61
5	QUANTITATIVE KODKOD	62
5.1	Atoms and Relations	62
5.2	Booleans Go Quantitative	64
5.3	Numeric Structures	66

5.4	Scope	69
5.5	Numeric Circuit Assembly	71
5.6	Summary	83
6	QUANTITATIVE ALLOY	84
6.1	Kodkod	84
6.2	Alloy and the Alloy Analyzer	97
6.3	Project Structure	108
6.4	Workflow	110
6.5	Quantitative Alloy in Practice	112
6.6	Summary	118
7	CASE STUDIES	120
7.1	Bibliometrics	120
7.2	Football Championship	125
7.3	Sprinkler	130
7.4	Bundling	138
7.5	Performance Results	141
7.6	Summary	147
8	CONCLUSION	149
8.1	Conclusions	149
8.2	Prospect for Future Work	153
A	LISTINGS	163
A.1	Bibliometrics	163
A.1.1	Alloy Model	163
A.1.2	Quantitative Alloy Model	163
A.1.3	Quantitative Instance	164
A.2	Football Championship	166
A.2.1	Alloy Model	166
A.2.2	SMT2-LIB Specification	167
A.2.3	Alloy Model using Quantitative Invariants	169
A.2.4	Quantitative Alloy Model with Quantitative Relations	171
A.2.5	Alloy Model with Multirelations	172
A.3	Sprinkler	173
A.3.1	Initial Scenario	173
A.3.2	Unknown Sprinkler	173
A.3.3	Alloy Model of a Bayesian Network	174

A.3.4	Bayes' Theorem in Alloy	175
A.3.5	Probabilistic Contract	177
A.4	Example of an Alloy Probabilistic Contract	179
A.5	Bundling	180
A.5.1	Alloy Specification using Multirelations	180
A.5.2	Quantitative Alloy Specification	181
A.5.3	Generalized Alloy Specification using Multirelations	182
A.5.4	Generalized Quantitative Alloy Specification	183
A.6	Benchmark	184

LIST OF FIGURES

Figure 2.1	Example of a relation and its respective kernel and image.	10
Figure 3.1	SMT-LIB logic fragments (Barrett et al., 2016).	33
Figure 3.2	Example of a relation on the left, and its <i>multirelation</i> representation on the right.	40
Figure 3.3	Q as the composition between S and R from Figure 3.2, portrayed as a span in the middle and as a relation whose arcs are weighted in the right.	42
Figure 3.4	Example of an Alloy specification dealing with <i>multiconcepts</i> .	43
Figure 3.5	One valid instance for the Alloy model in Figure 3.4.	43
Figure 3.6	Signature Σ_R of the theory T_R (Meng et al., 2017).	45
Figure 4.1	Example of the relation <i>History</i> .	50
Figure 4.2	Bayesian network for the Rain, sprinkler and grass problem.	55
Figure 4.3	Calculating the probability of the grass being wet using PRISM.	57
Figure 4.4	Constants range definition.	58
Figure 4.5	Verifying $P(g = 1) > 0.9$ with <i>experiments</i> .	58
Figure 4.6	Current Alloy Architecture.	60
Figure 4.7	Proposed Quantitative Alloy Architecture.	61
Figure 5.1	Declaring $R : A \rightarrow B$ in Alloy.	63
Figure 5.2	Specification of the unary set $S : A$ in a quantitative setting.	63
Figure 5.3	An instance from the model displayed in Figure 5.2.	64
Figure 5.4	A non-compact Boolean circuit on the left and the corresponding CBC ($d = 2$) in the right (Torlak and Jackson, 2007).	67
Figure 5.5	Extended Kodkod abstract syntax (adapted from (Torlak and Jackson, 2007)).	74
Figure 5.6	Application of the domain/range operator over a relation R described by its Kodkod matrix representation, within the universe A .	79
Figure 6.1	Generated PRISM model structure.	90
Figure 6.2	Txt representation of an Alloy instance.	106
Figure 6.3	Table representation of an Alloy instance.	107
Figure 6.4	Viz representation of an Alloy instance.	107
Figure 6.5	Quantitative Alloy Architecture.	110
Figure 6.6	Quantitative Alloy Workflow.	111
Figure 6.7	Quantitative Solving Options.	112

Figure 6.8	Example of the execution of a satisfiable command under the Integer context solved using an SMT Solver and an unsatisfiable result for a probabilistic model solved with PRISM.	113
Figure 6.9	Example of a command with and without imposing an integer scope.	114
Figure 6.10	Example of using the Alloy Evaluator to handle real valued numbers.	115
Figure 6.11	Instance determined for the constant f and δ considered.	117
Figure 6.12	Using the Alloy Evaluator to measure the probability of different contracts with respect to the instance's f and δ .	117
Figure 6.13	A δ determined by Alloy as a counterexample when checking the contract delimited by $p = \{a_1, a_2\}$ and $q = \{b_2\}$ over f for a probability lower than 80%.	118
Figure 6.14	The contract specified with $p = \{a_1, a_2\}$ and $q = \{b_2\}$ holds for the given f with, at least, 30% chance for every possible δ .	118
Figure 7.1	A quantitative instance of the original bibliographic system Alloy model.	121
Figure 7.2	The same Alloy solution (structure wise) when interpreted under Boolean versus quantitative semantics.	123
Figure 7.3	Example of a quantitative instance for the bibliographic system.	124
Figure 7.4	Number of citations each author received per area for the instance presented in Figure 7.3.	124
Figure 7.5	Potential instance of the bibliographic database under $Mat_{\mathbb{R}_0^+}$.	125
Figure 7.6	An instance produced after quantitative solving the Football Championship specification from Appendix A.2.1.	126
Figure 7.7	Checking if the alternative invariants associated with the requirement a) behave the same for a given set of participating teams and dates available.	128
Figure 7.8	A solution to the quantitative Alloy specification of the Football Championship.	129
Figure 7.9	Alloy instance representing a Bayesian Network for the given <i>rain</i> , <i>sprinkler</i> and <i>grass</i> .	131
Figure 7.10	Example of a possible <i>sprinkler</i> found by Alloy.	132
Figure 7.11	$P(g = 1)$ measured with respect to the presented network.	134
Figure 7.12	Finding a <i>sprinkler</i> for which $P(g = 1)$ is greater than 90%.	135
Figure 7.13	Calculating conditional probabilities through Bayes' theorem in Alloy.	136
Figure 7.14	Measuring $P(g = 1 \mid r = 1)$ using probabilistic contracts in Alloy.	137
Figure 7.15	Check if there is a δ for which $P(g = 1 \mid r = 1) \neq 80.19\%$.	138
Figure 7.16	SAT solver solution to the Bundling problem using <i>multirelations</i> .	139
Figure 7.17	Example of an instance provided by Quantitative Alloy for the Bundling model.	140
Figure 7.18	<i>multirelations</i> instance to the generalized bundling model.	141
Figure 7.19	Solution to the general bundling specification provided by the quantitative extension.	141
Figure 7.20	An instance of the Football Championship case study using <i>multirelations</i> .	144

LIST OF TABLES

Table 1.1	Category <i>Set</i> .	3
Table 1.2	Summary of the constructs of the ordered category <i>Rel</i> .	3
Table 1.3	Category <i>Mat</i> .	5
Table 2.1	Binary relation characterization (Oliveira, 2019).	10
Table 2.2	<i>Intersection</i> and <i>Union</i> of different kinds of matrix.	18
Table 2.3	Set operators supported by Alloy.	24
Table 2.4	Logical operators supported by Alloy.	25
Table 2.5	Alloy quantifiers.	25
Table 2.6	Multiplicity keywords.	26
Table 3.1	Expressions operators and functions supported by PRISM.	35
Table 3.2	Temporal operators.	36
Table 4.1	Example of a football championship obtained using CVC4.	54
Table 4.2	Extract of the experiment results.	58
Table 4.3	Parametric model checking results.	59
Table 5.1	Boolean gates supported by Kodkod.	72
Table 5.2	Newly added gates in the quantitative extension.	82
Table 6.1	Correspondence between the Numeric Structures and their respective SMT specification.	85
Table 6.1	Correspondence between the Numeric Structures and their respective SMT specification (continued).	86
Table 6.2	Translation of Numeric Structures into a PRISM model.	88
Table 6.2	Translation of Numeric Structures into a PRISM model (continued).	89
Table 6.3	Evaluating $\#(b <: S.Q) = 10$ with $Q = \{ y, z \}$.	113
Table 7.1	Bibliometrics – SAT versus SMT.	142
Table 7.2	Football Championship – SAT versus SMT.	143
Table 7.3	Quantitative adaptation of the Football Championship constraints.	143
Table 7.4	Football Championship with the quantitative relation <i>History</i> .	144
Table 7.5	Bundling example performance.	145
Table 7.6	Stats of solving the generalized bundling example.	146
Table 7.7	Finding different <i>sprinklers</i> .	146
Table 7.8	Testing Alloy when finding δ s for the probabilistic contract considered.	146

INTRODUCTION

With the evolution of technology, the ever-growing software capabilities reach more and more fields as time goes on. Each area of applicability possesses potentially intricate and distinct properties that require software-pieces tailored to them. Therefore, the variety and complexity of the software being built demands formal methods (ter Beek et al., 2019) general and expressive enough to meet their needs.

In particular, there has been a shift in interest from the well-established Boolean-valued analysis to further be able to measure and reason about quantities of multiple domains such as probabilities or time-based values, that is, the development of *quantitative formal methods* (Andova et al., 2009). These techniques work on a whole new range of systems whose intrinsic characteristics rely on such a quantification, where the usual *true* or *false* judgement lacks usefulness.

QUANTITATIVE METHODS Advances have been made regarding **probabilistic problems** such as quantitative safety properties verification combining probabilistic model checking and proof-based verification techniques (Ndukwu, 2009). More specifically, Andrews (2009) proposes stochastic extensions to the *Event-B* formal specification language and Bernardo (2009) describes a framework which allows testing of Markovian processes with internal exponential timed actions in polynomial time.

In the area of **real-time systems**, Wang and MacCaull (2009) extend a qualitative model checker to support real-time specification and verification, while AIAttili et al. (2009) solve scheduling problems using timed automatas with the *Uppaal* model checker. Moreover, li (2017) studies quantitative model checking of linear-time properties; Nigro and Sciammarella (2018) introduce a way of statistic model checking of distributed probabilistic timed actors using the *Theatre* modeling language. Quantitative reasoning emerges also in **quantum computing** in a rather obvious way, as the semantics of such programs are expressed by unitary matrices involving complex numbers, as can be seen in (Liu et al., 2019) and (Mateus and Sernadas, 2004). Even in the field of **natural language processing**, Baroni and Zamparelli (2010) take advantage of linear algebra to handle natural language semantics.

Despite the progress of this kind of approaches throughout the past years, the quantitative realm is still underdeveloped, due to the challenges involved in building new tools or enhance already existing ones to incorporate the new kinds of property, since the underlying theoretical framework needs to be sophisticated

and powerful enough to accommodate them. Moreover, tools are built around solving techniques that need to be refined, generalized or optimized, so that they can be used by practitioners. Last but not least, the increased complexity of the data being analysed may result in equally complex procedures to develop in order to manage them, which may hinder the tool's engine processing power and cause particularly slow response times, making it useless in the currently available machines.

Given the broad range of concepts that software deals with, one cannot help but wonder if there is a common way of reasoning about them, i.e., a theory sufficiently generic to approach all of them, while also still being specific enough so that their characteristics are perceptible and significant information can be extracted from them.

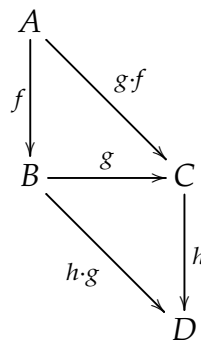
GOING GENERIC It turns out that **Category Theory** (MacLane, 1971) provides such a generic language. Through categories, one can reach multiple fields at once. A **category** \mathbb{C} is defined by a collection of *objects* which may or may not be involved in *arrows* – also called *morphisms* – coming *in* or *out* of them. Any two morphisms $f : A \rightarrow B$, $g : B \rightarrow C$ can be *composed* to form another morphism $g \cdot f : A \rightarrow C$. Then, \mathbb{C} represents a well-formed category as long as the following conditions are met:

- Every object A of \mathbb{C} contains an *identity morphism*:

$$\begin{array}{c} 1 \\ \curvearrowright \\ A \end{array}$$

such that $A \cdot 1 = A = 1 \cdot A$.

- Arrow composition is *associative*, meaning the following diagram commutes:



Therefore, $(h \cdot g) \cdot f = h \cdot (g \cdot f)$ for every object A, B, C, D and every arrow $f : A \rightarrow B$, $g : B \rightarrow C$, $h : C \rightarrow D$ in category \mathbb{C} .

The number of properties that can be expressed through simple “points” and “arrows” is not to be underestimated. For instance, a whole class of programs written in the so-called functional programming

paradigm can be represented and reasoned algebraically in the so-called category of sets Set (characterized in Table 1.1), whose objects are sets and whose morphisms are total functions.

Categorial	Functions	
$x \cdot y$	$f \cdot g$	Composition
1	id	Identity function

Table 1.1: Category Set .

Now, while one is able to calculate functional programs in such a setting, what if one needs to reason about the motivation behind them? That is, how could we specify their requirements and further determine if, in fact, the associated behaviour matches their purpose. Unfortunately, the category of sets seems to be lacking when trying to accomplish such a task, for instance, if one desires to constraint the program to process only a subset of its domain, ignoring all the inputs outside of the latter, it no longer can be modeled as a total function, but as a partial function instead, which do not belong in Set . Such as this last requirement, there are other kinds of properties which simply cannot be expressed by total functions and thus, the question arises: is there another ('better') category in which such a limitation could be overcome?

Since functions are a special cases of relations¹ one can try to resort to the category of binary relations Rel , with sets as objects and binary relations as morphisms. As it turns out, this category proves sufficient: after all, as functions form a subset of the whole relational universe, there is a big gain in expressiveness. This might come of no surprise given the success of Relational Databases in storing data from a huge variety of systems (solid IT, 2020). Besides, the morphisms in the Rel category can be ordered by inclusion, extending in a natural way the more traditional discipline of (homogeneous) *Relational Algebra*.

Categorial	Binary Relations	
$x \cdot y$	$R \cdot S$	Composition
$x + y$	$R \cup S$	Union
$x \times y$	$R \cap S$	Intersection
0	\perp	Empty relation
1	id	Identity relation
\top	\top	Top relation
x°	R°	Converse relation
$x \leq y$	$R \subseteq S$	Inclusion

Table 1.2: Summary of the constructs of the ordered category Rel .

It turns out that the abstract diagram notation provided by Category Theory allows for easy expression and flexible thinking of general concepts. Simply by perceiving the same diagram through different 'lenses' (categories) one can derive a whole set of fresh properties. Notably, the same construct which might be hard to deal within a certain category, can end up being quite simple in the eyes of another. Therefore, by

¹ Functions are exactly the so-called *simple* and *entire* binary relations (Oliveira, 2019).

cleverly picking the right categories we are able to increase expressiveness without any loss of previous knowledge (quite the opposite).

MODEL FINDING **Alloy**, a lightweight declarative specification language for software systems, is a successful case of a modeling tool based on first-order logic and relational algebra (Jackson, 2012). The simplicity of the language, together with the flexibility provided by relational reasoning makes Alloy very accessible to various types of users with various degrees of expertise. Basic understanding of discrete mathematics already enables one to design simple models and attain value from them, whilst not limiting more experienced users to build gradually more sophisticated models with increased familiarity of the language and its toolset. Notably, this toolset provides the **Alloy Analyzer**, whose features include ready-to-go automatic analysis of Alloy specifications, from instance finding to property checking by counterexamples that are displayed graphically and can be examined by the user through an interactive interface.

TOWARDS A QUANTITATIVE ALLOY Due to the expressive power of Alloy's underlying logic, its abstraction capabilities are very impressive when put up against complex systems. Unfortunately, along with the irrelevant information, the measurable data of the problem at hand also gets lost in the abstraction, meaning that, in its current state, Alloy is only able to perform simple qualitative analyses. Therefore, it is not very useful when dealing with problems with an inherent quantitative semantics, and this begs the question:

How could Alloy be augmented in order to support comprehensive quantitative analysis?

Put in other words: is there a category containing the same expressive power as *Rel* while, at the same time, being more and suitable for quantitative modelling and reasoning?

It can be easily observed that the cause of lost quantitative information in *Rel* arises from the *idempotency* of morphism addition, $x + x = x$, since relations are represented through sets and relational operators are closed under sets. Idempotency here means that one does not reach *multisets* when joining two relations, $R \cup R = R$. Therefore, to express quantitative information systems one should look for mathematical systems where $x + x > x$, for instance $x + x = 2x$.

Kodkod, a relational solver, is integrated beneath the Alloy Analyzer and is responsible for translating relational models into suitable propositional formulæ to be solved afterwards by an off-the-shelf SAT solver – a tool specially designed to determine the satisfiability of Boolean formulæ. Each relation in an Alloy specification is described in Kodkod by a **Boolean matrix**, meaning that if fact $b R a$ holds at Alloy level, then in the matrix cell identified by the a -column and b -row contains the value 1 at Kodkod level (and the value 0 otherwise). While the relational operators can be defined from the matrix representation point-of-view in a way that ensures closure under the universe of Boolean matrices, for instance specifying relational union² as $M \cup N = M + N - M \times N$, we are aware that matrix addition by itself is *not* idempotent.

² As long as M and N are both Boolean matrices, $M \cap N = M \times N$ and hence, it is intuitive that by adding them and then performing pointwise subtraction with their intersection, guarantees that their union is well-defined under Boolean values (Oliveira, 2012a).

Categorial	Matrices	
$x \cdot y$	$M \cdot N$	Matrix multiplication
$x + y$	$M + N$	Matrix addition
$x \times y$	$M \times N$	Hadamard product
0	\perp	Null matrix
1	id	Identity matrix
\top	\top	All-ones matrix
x°	M°	Transpose matrix
$x \leq y$	$M \leq N$	Semi-negative $M - N$

Table 1.3: Category *Mat*.

CATEGORIES OF MATRICES Let us then consider the category *Mat* whose morphisms are matrices and whose dimensions are the objects. Similarly to what happened before when we shifted from the *Set* category to *Rel*, now by going from *Rel* to *Mat* we are able to handle a whole new range of structures compared to the previous category, as Boolean matrices represent only a small subset of the entire matrix universe. Therefore, by releasing the “Boolean shackles” from the current relation representation, we might be able to reason about quantitative concepts.

In particular, what if instead of the Boolean-valued matrices, one considered the following alternatives:

- Category of *Numeric Matrices* $Mat_{\mathbb{N}_0}$

In place of $\{0, 1\}$ -valued elements, by allowing any natural number, one can reason in a more *Integer programming* approach. This change allows the characterization of relations whose arcs are weighted, and so $b R a = n$ specifies that a is related to b by R exactly n times. For example, *Book Own John = 4* can be interpreted as “John owns 4 books”, which could only be understood as “John owns *some* books” before, in the relation setting.

As exemplified above, through this representation one is able to implicitly handle numeric quantities in our specification. It should be noted that this characterization arose simply by perceiving the same relation from a different light. There was no need to add new constraints, just to change the category (Oliveira and Miraldo, 2016).

However, once one is dealing with numeric values, the usual SAT solvers lack expressiveness to be able to solve quantitative constraints. Thus one must take advantage of SMT solvers, automatic solvers which are able to check the satisfiability of a first-order logic formula with respect to a specific background theory (Frade, 2019c).

- Category of *Left-Stochastic Matrices* *LS*

By considering Left-stochastic matrices – matrices whose elements are probabilities, i.e., real numbers in the interval $[0, 1]$ and whose columns all add up to 1 – one is able to represent and analyse

probabilistic structures like distributions and conditional probability tables, for instance. Moreover, with f describing a probabilistic function (whose behaviour is to yield a distribution of outputs rather than a single output), one may introduce the notion of **probabilistic contract** $q \xleftarrow{f} p$ with p specifying the *precondition* and q the *postcondition*, and then be able to model-check the contract according to a potentially unknown distribution δ , allowing us to quantify defective performance in software.

Moreover, introducing probabilistic thinking changes the subject from “may it happen?” to “*how often* will it happen?” and thus, we are now interested in performing *Performance Risk Analysis* of software systems:

1. What can go wrong?
2. How likely is it?
3. What are the consequences?

preferably by taking advantage of tools capable of automatically find answers to all these questions.

Given the nature of the values that we are now dealing with, instead of SAT solvers one must take advantage of a probabilistic or statistical model checkers in order to handle this type of constraints.

Initially, we went from functional behaviour (determinism) to relations (non-determinism), and now, by lifting relations to matrices we are able to reason about quantities (probabilism). During this progress, the expressiveness at each point increased progressively and without the need of explicitly introducing new notions, and thus, we followed the *scalable modeling* lemma: “keep definition, change category” (Oliveira and Miraldo, 2016). Indeed, through different categories we were able to reason about the same concepts on completely different levels, each with its own advantages and disadvantages. With this dissertation we hope to extend Alloy’s underlying theoretical framework through this line of thought, providing support to both $Mat_{\mathbb{N}_0}$ and LS concepts in a *typed linear algebra* fashion. Furthermore, we wish to implement them in the Alloy Analyzer, which implies designing (or finding) an adequate theory for $Mat_{\mathbb{N}_0}$ so that the new type of quantitative constraints can be handled by an off-the-shelf SMT solver; and integrate a suitable probabilistic or statistical model checker to reason over LS . In the end, we hope to strengthen Alloy’s capabilities as an already outstanding modelling tool, broadening the range of problems it can tackle, promoting the progress of quantitative formal methods.

1.1 STRUCTURE OF THE DISSERTATION

Chapter 2 of this dissertation starts by presenting the theoretical foundations considered for this work, followed by the characterization of the main tools involved in the project and the study of existing research relevant to the subject at hand in Chapter 3. Next, Chapter 4 will introduce the case studies for which we

wish to apply the results of the project's development and explore how we can currently solve them using state-of-the-art tools; then, we propose an initial architecture design of the solution that will be produced. Chapter 5 goes over the implementation decisions and Chapter 6 puts them into practice, detailing the extension developed in-depth and presenting the finished tool architecture and workflow. Afterwards, Chapter 7 appraises the quantitative extension of Alloy when put up against the case studies presented in Chapter 4, together with a benchmark to measure its performance. Finally, Chapter 8 examines all the work made throughout this dissertation, evaluates the degree of accomplishment of the initial objectives and proposes the next steps to improve this project.

 BACKGROUND

As mentioned previously, Alloy's fundamental logic is based on *Relational Algebra*, which includes a number of relational operators highlighted for *Rel* in Table 1.2, alongside other useful relational concepts. The semantics of such operators and concepts can be expressed in the so-called *Eindhoven quantifier* notation, where expressions of the form $\langle \forall x : P : Q \rangle$ (resp. $\langle \exists x : P : Q \rangle$) state that, for *all* (resp. *some*) x in the range P , Q holds (Oliveira, 2019). This chapter addresses such a background in a succinct way. It is worth mentioning that the logic at hand does not strictly follow traditional Relational Algebra, for instance, while in the following section the theoretical background will be presented under *Rel* and so, the concepts will be approached over *binary* relations, as usual in the context of relational algebra, Alloy also supports relations of greater arity – which in turn conform to a generalized version of the theory to be presented –. Further differences between the two can be pointed out, deliberately put into place to promote the language flexibility/convenience (e.g. its type system allows the application of an operator between two sets whose types do not match, within a certain degree of freedom, and so on).

Different kinds of *decision problems* and *probabilistic model checking* concepts will be hereafter focused on, especially those associated with the theoretical component of Alloy and the other tools analysed in the next chapter, namely the *Boolean satisfiability problem*, which is currently used in Alloy to verify the model's properties over Boolean matrices, and the introduction of the *satisfiability modulo theory problem* to handle numeric matrices and the constraints that they will be subject to in the quantitative extension.

To conclude, this chapter goes over Alloy itself, detailing its models' structure, language, features and all.

2.1 FIRST-ORDER RELATIONAL LOGIC

Let A and B be types. A binary **relation** $A \xrightarrow{R} B$ describes an association between inhabitants of A to inhabitants of B – i.e. R is *well-typed* on $A \rightarrow B$ – so that (a, b) is an element of R iff $a \in A$ is related to $b \in B$ under R . For instance, if $Item \xleftarrow{Own} Person$ describes which kind of items are owned by some person, with $Person = \{John\}$, $Item = \{Book, Pen, Pencil\}$ and $Own = \{(John, Book), (John, Pen)\}$ then, Own specifies that John possesses book(s) and pen(s), but no pencils.

Binary relations are the morphisms of a category usually termed *Rel*. Notations $A \rightarrow B$ or $B \rightarrow A$ are interchangeable. The **identity relation** on a type A is denoted by $A \xleftarrow{id} A$, cf.:

$$id_A = \{(a, a) \mid a \in A\} \quad (2.1)$$

$$R \cdot id_A = R = id_B \cdot R \quad (2.2)$$

It should be pointed out that as long as it is not ambiguous, i.e., which set A is associated to id_A is implied by the context, in general the subscript is dropped and simply the notation id is adopted.

Moreover, any two relations with composable types can be combined to create a new relation:

$$b(R \cdot S)c = \langle \exists a : b R a : a S c \rangle \quad (2.3)$$

additionally, relational **composition** must be *associative*,

$$(R \cdot S) \cdot Q = R \cdot (S \cdot Q) \quad (2.4)$$

for every $R : A \rightarrow B, S : C \rightarrow A, Q : D \rightarrow C$.

Unlike functions, every relation R gets a **converse** relation R° ,

$$b R a \Leftrightarrow a R^\circ b \quad (2.5)$$

Relation **inclusion** $R \subseteq S$ holds if and only if every element of R is also in relation S ,

$$R \subseteq S \equiv \langle \forall a, b :: b R a \Rightarrow b S a \rangle \quad (2.6)$$

Given any relation $R : A \rightarrow B$, one can reason about its domain and codomain through its **kernel** and **image**, respectively:

KERNEL $\ker R = R^\circ \cdot R$

IMAGE $\text{img } R = R \cdot R^\circ$

The kernel (resp. image) of R is the relation which associates two sources (resp. targets) given that they share at least one common target (resp. source), as illustrated in Figure 2.1. With the help of these concepts, a number of properties can be derived about the relation at hand, in particular by comparison with the identity relation, as shown in Table 2.1.

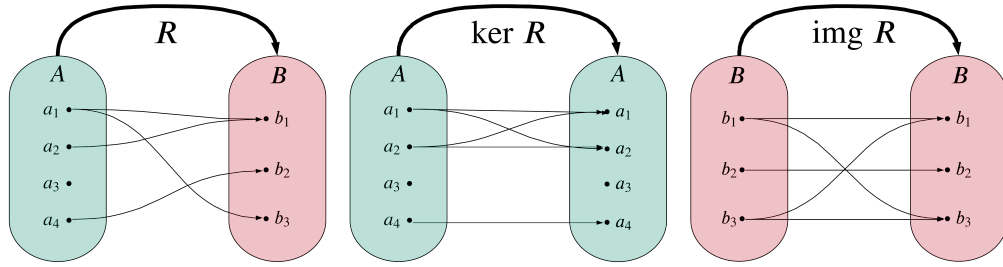


Figure 2.1: Example of a relation and its respective kernel and image.

X	$id \subseteq X$	$X \subseteq id$
$\ker R$	total R	injective R
$\text{img } R$	surjective R	simple R

Table 2.1: Binary relation characterization (Oliveira, 2019).

Furthermore, by checking that $\text{img } R \subseteq id$ and $id \subseteq \ker R$, it is possible to conclude that R is simple and entire, and thus, R describes a **function**. Functions are written lowercase, e.g. f, g, \dots . In case of functions, notation $b f a$ means $b = f a$.

Let R be the relation presented in Figure 2.1. By observing its kernel and image as presented above, one can see that only $id \subseteq \text{img } R$ holds, meaning that this relation is surjective, but is neither total, injective or simple (which can be further confirmed by looking at the R pictured).

Two relations of the same type can be combined through:

MEET $R \cap S$ which defines the relation resulting from the intersection of R and S .

$$b (R \cap S) a \equiv b R a \wedge b S a \tag{2.7}$$

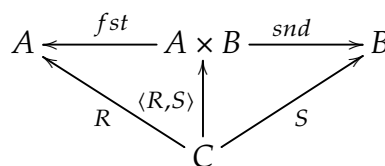
JOIN $R \cup S$ corresponds to the union of relations R and S .

$$b (R \cup S) a \equiv b R a \vee b S a \tag{2.8}$$

By definition, one can show that the \cup -idempotency holds.

When dealing with relations with potentially different types, they can be connected by taking advantage of:

PRODUCT Every relation $R : C \rightarrow A, S : C \rightarrow B$ can be *paired* to form $\langle R, S \rangle$ as follows,



$$(a, b) \langle R, S \rangle c \Leftrightarrow a R c \wedge b S c \tag{2.9}$$

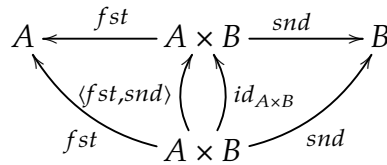
where $fst : A \times B \rightarrow A$ and $snd : A \times B \rightarrow B$ are *projections* which act as expected,

$$fst(a, b) = a \wedge snd(a, b) = b \tag{2.10}$$

Then, in general, the **product** $Q \times P : A \times B \rightarrow C \times D$ is defined as

$$Q \times P = \langle Q \cdot fst, P \cdot snd \rangle \tag{2.11}$$

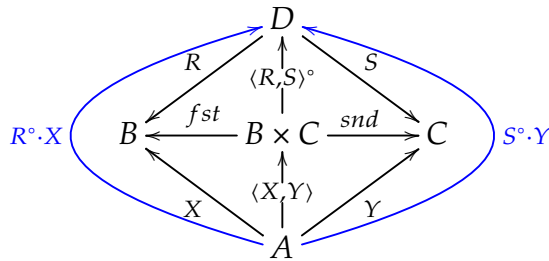
By taking R and S as the product projections, the pairing diagram is now the following:



describing what is called the **x-reflection** law:

$$\langle fst, snd \rangle = id_{A \times B} \tag{2.12}$$

Additionally, relational pairing has an useful property:



$$\langle R, S \rangle^\circ \cdot \langle X, Y \rangle = (R^\circ \cdot X) \cap (S^\circ \cdot Y) \tag{2.13}$$

from which one is able to relate some of the concepts introduced above, as follows:

$$\begin{aligned} & \ker \langle R, S \rangle \\ &= \{ \text{Definition of kernel} \} \\ & \langle R, S \rangle^\circ \cdot \langle R, S \rangle \\ &= \{ (2.13) \} \\ & (R^\circ \cdot R) \cap (S^\circ \cdot S) \end{aligned}$$

$$= \{ \text{Definition of kernel} \}$$

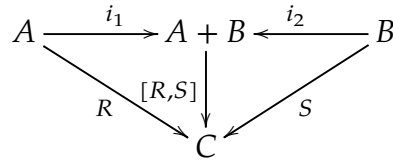
$$\ker R \cap \ker S$$

Thus,

$$\ker \langle R, S \rangle = \ker R \cap \ker S \tag{2.14}$$

an interesting property which will be made use of in a future chapter.

CO-PRODUCT The dual constructs are analogously characterized by “inverting arrows” and so, the *either* of any relations $R : C \leftarrow A$ and $S : C \leftarrow B$ can be specified with $[R, S]$,



$$[R, S] = R \cdot i_1^\circ \cup S \cdot i_2^\circ \tag{2.15}$$

where i_1 and i_2 are the *injections* of the disjoint union defined as,

$$i_1 a = (t_1, a) \wedge i_2 b = (t_2, b) \tag{2.16}$$

Therefore, the **coproduct** $Q + P : C + D \leftarrow A + B$ is given by

$$Q + P = [i_1 \cdot Q, i_2 \cdot P] \tag{2.17}$$

EXPONENTIAL Given two sets A and B , an **exponential** B^A is the type of all functions from A to B .

$$B^A = \{f \mid f : A \rightarrow B\} \tag{2.18}$$

The **transitive closure** \mathcal{R} of an *endo-relation* $R : A \rightarrow A$ is characterized as the smallest relation satisfying the following

$$\mathcal{R} = R \cup R \cdot R \cup R \cdot R \cdot R \cup \dots \tag{2.19}$$

Each relational type $A \rightarrow B$ has the following special relations:

- **Top** relation $T : A \rightarrow B$, which represents the *Universal* relation

$$b T a \equiv true \tag{2.20}$$

- **Bottom** relation $\perp : A \rightarrow B$, also known as the *Empty* relation,

$$b \perp a \equiv \text{false} \quad (2.21)$$

The notation for each universal (resp. empty) relation is overloaded when the type of the top (resp. bottom) at hand can be inferred, being represented by the same symbol presented above.

Alongside id_A , any object A in the *Rel* category is uniquely associated to the constant function $!_A : A \rightarrow 1$, where 1 denotes the singleton object. This function is usually referred to as the **Bang** function. It sends every element of A to the *unit* value, i.e. the sole inhabitant of 1 which can be interpreted as the *null-pointer* from the imperative paradigm point-of-view. Analogously to the id_A notation, the subscript of bang will also be omitted when its type is implicit from the context. Clearly, every function of type $1 \rightarrow A$ (for A nonempty) is also a constant function, known as a *point*. As there are as many such points as $a \in A$, we write $\underline{a} : 1 \rightarrow A$ to indicate the particular choice of constant function.

One can define a **functional contract** $q \xleftarrow{f} p$ for some function $f : A \rightarrow B$ over a *precondition* $p : A \rightarrow \mathbb{B}$ and a *postcondition* $q : B \rightarrow \mathbb{B}$, constraining the inputs of f to those that pass the precondition, guaranteeing that, for those inputs, f only produces outputs for which the postcondition holds,

$$q \xleftarrow{f} p = \langle \forall a : p a : q (f a) \rangle \quad (2.22)$$

The notion of contract can be generalized to relations such that, a **relational contract** $q \xleftarrow{R} p$ over $R : A \rightarrow B$ with respect to the conditions $p : A \rightarrow \mathbb{B}$ and $q : B \rightarrow \mathbb{B}$ specifies that, as long as the inputs satisfy p , then it is known for sure that q holds for the outputs that are related to the inputs under R , i.e.,

$$q \xleftarrow{R} p = \langle \forall a : p a : \langle \forall b : b R a : q b \rangle \rangle \quad (2.23)$$

2.2 TYPED LINEAR ALGEBRA

Here is presented the shift from *Rel* to *Mat*, which includes describing the concepts studied before from the category of matrices point-of-view, i.e., the operators presented in Table 1.3 together with the newly acquired capabilities, given the increase in expressiveness through [Typed Linear Algebra](#).

As stated previously, every relation can be represented through a **Boolean Matrix** – a matrix where each element is $\{0, 1\}$ -valued – as shown in the following example:

$$\begin{aligned} \{b_1, b_2, b_3\} &\xleftarrow{R} \{a_1, a_2, a_3\} =_{Rel} \{(a_1, b_2), (a_1, b_3), (a_2, b_1), (a_2, b_2)\} \\ &=_{Mat} \begin{matrix} & a_1 & a_2 & a_3 \\ b_1 & \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \\ b_2 & \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \\ b_3 & \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \end{matrix} = 3 \xleftarrow{R} 3 \end{aligned}$$

Furthermore, M represents a Boolean matrix as long as $M \times M = M$, where \times corresponds to *Hadamard product*, since $0 \times 0 = 0$ and $1 \times 1 = 1$ ($n \times n \neq n$ for any other number).

Naturally, given that Mat is a category, then there is an **identity matrix** for every dimension and arrow composition corresponds to **matrix multiplication**, which is possible if and only if the number of columns of the first matrix is equal to the number of rows of the second – the type of a matrix corresponds to its dimensions –, both well-defined as follows:

$$n \xleftarrow{id_n} n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} \tag{2.24}$$

$$n \xleftarrow[M]{M \cdot N} m \xleftarrow[N]{N} k \quad b(M \cdot N)c = \langle \sum a :: b M a \times a N c \rangle \tag{2.25}$$

Vectors can also be represented, as they are matrices where one of its dimensions is 1:

- On one hand, a *row vector* $1 \xleftarrow{r} m$ has m columns and one row;
- On the other hand, a *column vector* is of the form $n \xleftarrow{c} 1$ having one column and n rows;
- Every vector of type $1 \rightarrow 1$ is known as a *scalar*, usually denoted by the cell it contains.

The matrix representation of $\text{Bang} !_m : m \rightarrow 1$ is row vector where every cell has value 1.

In particular, a matrix f corresponds to a function if it is a Boolean matrix such that $! \cdot f = !$.

The **converse** of a matrix $n \xleftarrow{M} m$ is defined as the *transpose* of that matrix, which swaps the row and column position from its elements, $m \xleftarrow{M^o} n$.

As stated previously in the introduction, *+idempotency* is overcome by considering **matrix addition**. Any two matrices can be added as long they have the same type, $M + N : n \leftarrow m$ is the resulting matrix

from performing pointwise addition of $M, N : n \leftarrow m$. Moreover, composition is **bilinear** with respect to $+$ and thus,

$$M \cdot (N + P) = M \cdot N + M \cdot P \quad (2.26)$$

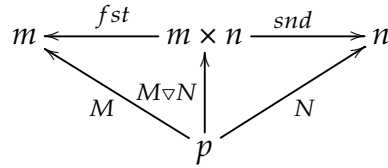
$$(M + N) \cdot P = M \cdot P + N \cdot P \quad (2.27)$$

Analogously to *Rel*, in *Mat*, for every matrix dimensions $m \rightarrow n$ there are matching **top** and **bottom** matrices, defined as:

- $\top : m \rightarrow n$ is the matrix where every element is 1, and so, *bang* is a special case of top for row vectors;
- $\perp : m \rightarrow n$ corresponds to the *null-matrix*, i.e., a matrix filled with 0s.

Mat also provides product and coproduct constructs:

PRODUCT The *pairing* of two matrices $M : p \rightarrow m$, $N : p \rightarrow n$ can be achieved through *Khatri-Rao product* $M \nabla N : p \rightarrow m \times n$ such that,



$$(b, c) (M \nabla N) a = (b M a) \times (c N a) \quad (2.28)$$

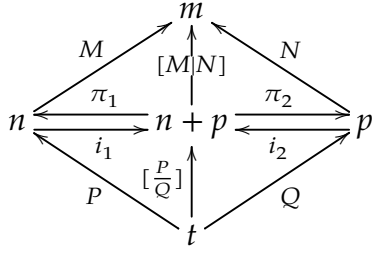
$$fst \cdot (M \nabla N) = M \wedge snd \cdot (M \nabla N) = N \quad (2.29)$$

where $fst : m \times n \rightarrow m$ and $snd : m \times n \rightarrow n$ are the corresponding product projections.

COPRODUCT In this context, coproduct provides two polymorphic combinators over matrix **block notation**:

- $[M \mid N] : n + p \rightarrow m$ – the *junc* combinator associates $M : n \rightarrow m$ and $N : p \rightarrow m$ horizontally;
- $\left[\begin{array}{c} P \\ Q \end{array} \right] : n + p \leftarrow t$ – *split* stacks $P : t \rightarrow n$ on top of $Q : t \rightarrow p$.

defined by:



$$[M \mid N] = M \cdot \pi_1 + N \cdot \pi_2 \quad (2.30)$$

$$\left[\begin{array}{c} P \\ Q \end{array} \right] = i_1 \cdot P + i_2 \cdot Q \quad (2.31)$$

where the projection π_1, π_2 and the injections i_1, i_2 for coproducts obey to the following:

$$\left[\begin{array}{c} \pi_1 \\ \pi_2 \end{array} \right] = id \quad (2.32)$$

$$[i_1 \mid i_2] = id \quad (2.33)$$

$$i_1 = \pi_1^\circ \wedge i_2 = \pi_2^\circ \quad (2.34)$$

TYPED MATRICES Matrix types can be **generalized** to denumerable sets like A, B , similarly to relational types. For instance, $A + B$ (the disjoint union A and B) will replace the addition of matrix dimensions $n + m$. Therefore, a new definition to the **matrix transformation** $\llbracket R \rrbracket$ can be provided: any relation $R : A \rightarrow B$ has a corresponding Boolean matrix representation $M : A \rightarrow B$ so that:

$$M = \llbracket R \rrbracket \equiv \langle \forall a, b :: b M a = \mathbf{if} (b R a) \mathbf{then} 1 \mathbf{else} 0 \rangle \quad (2.35)$$

Stochastic matrices are a good illustration of the typed matrix concept. The category of Left-Stochastic matrices LS , i.e., matrices with type $A \rightarrow B$ whose cells are valued over the unit interval $[0, 1] \subseteq \mathbb{R}^1$ enables algebraic reasoning about probabilistic notions. As indicated initially, the Alloy extension should be able to reason about **probabilistic contracts** (Oliveira, 2017a) and thus, the following essential probabilistic concepts are highlighted:

EVENT An **event** $e : A \rightarrow 1$ corresponds to a Boolean row vector which describes a deterministic instance over A , meaning that for every $a \in A$, if $e \cdot \underline{a} = 1$ then a occurs, and it does not in case $e \cdot \underline{a} = 0$.

PROBABILISTIC FUNCTION The chance of a specific $b \in B$ being associated with a given input is precised described by a distribution μ . In general, the family of distributions over B is referred to as \mathcal{DB} ,

$$\mathcal{DB} = \{ \mu \in [0, 1]^B \mid \sum_{b \in B} \mu b = 1 \} \quad (2.36)$$

1 Recall page 5.

A *probabilistic function* f from A to B relates every value of A to each value of B with respect to some probability, i.e., $(f a)b = p$ (or $b =_p f a$) describes that the probability of f returning $b \in B$ for a given $a \in A$ is equal to $p \in [0, 1]$.

Therefore, f captures the chance of b being related to a value a and thus, the type of a probabilistic function includes all distributions on B , so, $f : A \rightarrow \mathcal{D}B$.

MATRIX TRANSFORM The *left-stochastic* matrix $M : A \rightarrow B$ associated to some probabilistic function $f : A \rightarrow \mathcal{D}B$ is obtained by the *matrix transformation* $\llbracket \cdot \rrbracket$, analogously to the transformation between relations and Boolean matrices,

$$M = \llbracket f \rrbracket \equiv \langle \forall a, b :: b M a = (f a)b \rangle \tag{2.37}$$

meaning that the element at the position (a, b) of M identifies the probability of f outputting b for the input a .

DISTRIBUTION $\delta : 1 \rightarrow A$ denotes a *distribution vector* over a probabilistic event set A , subject to:

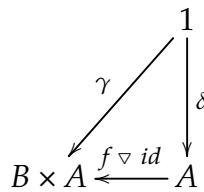
$$\langle \sum a :: a \delta _ \rangle = 1$$

CONDITIONAL PROBABILITY Given two events $a : A \rightarrow 1$ and $b : B \rightarrow 1$, the conditional probability $P_\delta(a | b)$ of a happening assuming that b occurs over some distribution δ is given by:

$$P_\delta(a | b) = \frac{(a \times b) \cdot \delta}{b \cdot \delta} \tag{2.38}$$

PROBABILISTIC CONTRACT For any probabilistic function $f : A \rightarrow \mathcal{D}B$ and predicates $p : A \rightarrow 1$, $q : B \rightarrow 1$, the associated *probabilistic contract* $q \xleftarrow{f} p$ holds different probabilities for different input distributions $\delta : 1 \rightarrow A$. Hence, in order to check a probabilistic contract, one must find which input values violate the contract over δ .

After fixing some δ , the distribution γ – which captures the probabilistic behaviour of f according to that input distribution – must be determined,



$$\gamma = (f \triangleright id) \cdot \delta \tag{2.39}$$

In order to measure the probability of the contract holding for that distribution, the conditional probability of the values produced by f satisfy the postcondition q , given that its input are valid under the precondition p , is evaluated with respect to the input distribution δ :

$$\llbracket q \xleftarrow{f} p \rrbracket_{\delta} = P_{\gamma}(q \cdot fst \mid p \cdot snd) \quad (2.40)$$

which further simplifies into

$$\llbracket q \xleftarrow{f} p \rrbracket_{\delta} = (q \cdot f \times p) \cdot \frac{\delta}{p \cdot \delta} \quad (2.41)$$

Lastly, another useful probabilistic operator corresponds to **probabilistic choice** from the $pCGL$ programming language (Gretz et al., 2015) $\{P\}[p]\{Q\}$ which executes the program P with probability p and Q with probability $(1 - p)$. In this setting, probabilistic choice can be encoded as selecting one of two probabilistic functions f, g of the same type, following the same line of thought, denoted $\llbracket f \diamond_p g \rrbracket$.

OPERATORS PER MATRIX KIND Unlike the constructs presented previously in general for any kind of matrix under Mat , there are some concepts from Rel which cannot be characterized in such generic terms for every type of matrix. Thus, those operators will now be defined for each kind of matrix in focus throughout this dissertation specifically, so that closure under their domain is guaranteed.

Any two matrices with the same type M, N can be combined through **intersection** and **union** as defined in Table 2.2.

	Intersection		Union	
	$M \cap N$		$M \cup N$	
$Mat_{\mathbb{B}}$	$M \times N$	(2.42)	$M + N - M \times N$	(2.43)
$Mat_{\mathbb{N}_0}$	$\min(M, N)$	(2.44)	$M + N$	(2.45)
LS	$\text{norm}(\min(M, N))$	(2.46)	$\text{norm}(M + N)$	(2.47)

Table 2.2: *Intersection* and *Union* of different kinds of matrix.

$\text{norm}(D)$ ensures that $D : A \rightarrow B$ is a proper left-stochastic matrix, that is: for each distribution δ_i associated with every column $i \in A$ of D ,

$$\text{norm}(\delta_i) = \frac{\delta_i}{\sum_{j \in B} j \delta_i} \quad (2.48)$$

Transitive closure can also be defined for any *square* Boolean matrix $M : n \rightarrow n$, similarly to the transitive closure of an endo-relation,

$$\tilde{M} = M \cup M \cdot M \cup M \cdot M \cdot M \cup \dots \quad (2.49)$$

2.3 BOOLEAN SATISFIABILITY PROBLEM

A **decision problem** is a question that can be answered with “yes” or “no” depending on the input provided (Frade, 2019a). A decision problem is said to be **decidable** if there exists a procedure capable of always *terminating* and producing an answer for *any* admissible input.

The **Boolean Satisfiability Problem** (also known as **SAT**) is an example of a decidable decision problem for *propositional logic formulas*. A formula \mathcal{F} is **satisfiable** if there is at least one *assignment* A – a map from each logical variable of \mathcal{F} to either *true* or *false* – for which the formula is *true*, meaning that A *models* \mathcal{F} , written as $A \models \mathcal{F}$. If \mathcal{F} is **unsatisfiable**, then there are no assignments that make \mathcal{F} hold and thus, \mathcal{F} represents a *contradiction*, denoted as $\not\models \mathcal{F}$.

Tools specifically built to solve this problem are called **SAT solvers** (Gu et al., 1997; Cook and Mitchell, 2000; Kilani et al., 2013), which return *sat* and an assignment in case the given formula is satisfiable or *unsat* together with a proof otherwise. These solvers typically receive as input formulas in their respective **conjunctive normal form** (also known as **CNF**), that is, a formula of the form:

$$\bigwedge_i (\bigvee_j l_{ij})$$

where l_{ij} is the j -th **literal** – a proposition of the form p or $\neg p$, where p represents an atomic predicate – in the i -th clause. Notably, every propositional formula has an equivalent CNF formula representation.

While the SAT problem is decidable, it is also NP-complete, hence effective procedures are needed to handle formulas with a massive number of variables and clauses. However, state-of-the-art SAT solvers implement sophisticated and efficient techniques, making them able to solve such formulas in an acceptable amount of time. Most modern SAT solvers have implemented **DPLL**-based procedures, which include backtrack search and traversing on a binary tree representing the input formula, as it is the conventional current most effective approach.

MiniSAT (Eén and Sörensson, 2004) and SAT4J (Le Berre and Parrain, 2010) are some examples of SAT solvers.

2.4 SATISFIABILITY MODULO THEORIES

Going from propositional logic into *first-order logic* results in the increase of expressiveness, as the language provides not only the usual logical operators, but also quantifiers \exists and \forall , variables, constants and all (Frade, 2019b).

Similarly to propositional logic formulas, there are decision problems variations for first-order logic formulas. In particular, given a FOL formula ϕ , a \mathcal{U} -structure \mathcal{M} and an assignment α :

VALIDITY ϕ is **valid** if and only if $\mathcal{M}, \alpha \models \phi$ holds for *any* structure \mathcal{M} and *any* assignments α , in which case ϕ is a *tautology*, written as $\models \phi$.

SATISFIABILITY ϕ is **satisfiable** if there exists at least one structure \mathcal{M} and assignment α where $\mathcal{M}, \alpha \models \phi$.

Yet, unlike SAT for propositional logic, the validity and satisfiability problems are both **undecidable** for first-order logic, meaning that there is no procedure that is able to always terminate and give a legitimate “yes” or “no” judgement to all of the admissible FOL formulas as input. In regards to the FOL supported by Alloy, the decision problem associated is decidable, capable of being encoded and further solved through SAT, due to the fact that the framework reasons over a finite universe, while for first-order logic generally this is not the case, as an infinite domain is commonly considered, therefore motivating the need for this class of decision problems.

Nevertheless, these decision problems are **semi-decidable**, meaning there exists a method for each of them that, given any first-order logic formula, is able to:

- Terminate and produce the answer “yes” iff it is the correct answer;
- Halt and return “no” when “no” is the legitimate answer;
- Or it does not terminate if “no” corresponds to the correct result.

While, in general, first-order logic validity/satisfiability problem is undecidable, there are decidable **fragments** of the logic.

A popular variant of FOL in software modeling corresponds to **Many-sorted first-order logic**, which extends the first to the concept of **sorts** (or types of objects), providing access to all the FOL properties alongside constraints on operations and predicates over different kinds of sorts, due to them being distinguishable at the syntactical level.

When checking the validity/satisfiability of a given formula, one is usually interested in doing so relatively to some specific context, for instance, consider the following formula over integers:

$$\forall x. x \times x \geq 0$$

it is not interesting to verify if such formula is valid/satisfiable for *all* interpretations of \times , but only those where \times represents the usual integer multiplication operator. Therefore, one should consider validation with respect to a **background theory**.

A first-order **theory** \mathcal{T} over the logic language provides a set of axioms which outline the class of models that will be considered during the judgement, such that, a formula $\phi \in \mathcal{T}$ is:

- \mathcal{T} -*valid* if every \mathcal{T} -structure validates ϕ ;
- \mathcal{T} -*satisfiable* as long as there is at least one \mathcal{T} -structure which validates ϕ .

with \mathcal{T} -*structure* representing a \mathcal{U} -structure where every formula of \mathcal{T} is validated. Some instances of well-established theories include the theory of *Peano arithmetic* \mathcal{T}_{PA} and the theory of *Linear integer arithmetic* \mathcal{T}_Z .

In conclusion, the **Satisfiability Modulo Theories** problem (also called **SMT**) is the decision problem over the satisfiability of a first-order logic formula with respect to a background theory \mathcal{T} . The tools built to handle such problem are known as **SMT solvers** (Monniaux, 2016), which typically reason over formulas specified in many-sorted FOL, passing the *sat*, *unsat* or *unknown* (timeout, out of memory, ...) judgement according to a given theory.

2.5 PROBABILISTIC MODEL CHECKING

The automatic verification technique known as model checking can be expanded to handle probabilistic settings, giving origin to **Probabilistic Model Checking** (Parker, 2011). This type of formal method arises from the need of modeling and verifying systems whose characteristics are intrinsically probabilistic, displaying random behaviour in a probabilistic or non-deterministic manner and thus, requiring quantitative analysis beyond the usual qualitative judgement.

PROBABILISTIC MODELS From the many existing types of probabilistic models supported in probabilistic model checking tools, the following can be highlighted as the most common:

DISCRETE-TIME MARKOV CHAINS A DTMC describes probabilistic state-transition systems. Its *states* represent every possible configuration that the system might be in at some point; the change between configurations occurs through *transitions* that happen in *discrete-time steps*; the shift between states follows a *discrete probability distribution*.

CONTINUOUS-TIME MARKOV CHAINS Unlike DTMC, the state transitions of continuous-time Markov chain do not occur in discrete-time units, as it corresponds to a dense model of time where a transition can happen at any point in time. The time instants are modelled through exponential distributions.

This type of model is useful to study scenarios regarding failures, arrival times, and so on.

MARKOV DECISION PROCESSES A Markov decision process is a nondeterministic extension to the DTMC, adequate to handle systems that features both probabilistic and nondeterministic characteristics.

In particular, the nondeterministic capabilities allow specification of concurrency and unknown environments for instance.

Before handling probabilistic behaviour in a MDP, first the nondeterministic choices must be made. For that, one resorts to **Adversaries** (also called **Schedulers**), which are responsible to make the nondeterministic decisions. These adversaries can have different characteristics and be of different kinds, their choice might be *memoryless*, *random*, *fair* or of some other type.

Every single one of these models possesses the **Markov property**, also known as **memorylessness**, which specifies that if the current configuration is known, then the future states of the system are independent of its previous configurations, meaning that current state of the model contains all the information that can impact the future changes in the system configuration.

Furthermore, these probabilistic models may or may not have **costs** and **rewards** associated to its states and transitions. Costs and rewards correspond to real-valued quantities that are formally indistinguishable, their interpretation depends case by case. By taking advantage of them, one can quantify useful attributes related to the system at hand, like elapsed time, size, expenses and others. They can also be used in property specification as will be shown next.

PROBABILISTIC LOGIC In order to specify properties over a probabilistic model, one of multiple (probabilistic) temporal logics can be used, like the probabilistic version of computation tree logic (PCTL), (probabilistic) LTL, continuous stochastic logic (CSL) and others. Through these logics one can express various kinds of properties depending on the type of model that is being considered:

- **Steady-state behaviour** analyses the state of the system in the long-run;
- **Probabilistic (Repeated) Reachability** reasons about the probability of reaching a certain state (infinitely many times in the repeated reachability case);
- **Probabilistic Invariance** describes the probability of the system remaining in a subset of all possible states;
- **Persistent properties** examine the probability of a system ending up in the same state endlessly (terminal state/deadlock);
- **Minimum and Maximum probabilities** of reaching a target set of states within a MDP;
- ...

The above-mentioned type of properties can be specified for both qualitative (being *true* or *false* for a specific model) and quantitative reasoning (measuring the probability with respect to the model at hand).

If the model has rewards (or costs) associated, *Reward-based properties* can be specified:

- **Instantaneous** reason over the *expected* value of the reward in a specific state;
- **Cumulative** considers the *expected* cumulative value of the rewards, potentially up to a specific time instance;
- **Reachability** analyses the possibility of the reward reaching an expected cumulative value before the system goes into some state.

Likewise, these types of properties can also be subject to qualitative or quantitative verification.

2.6 ALLOY

The Alloy specification language, alongside the Alloy Analyzer, will be the main focus during the project. Having its foundation in relational algebra, this simple, flexible and lightweight modeling language turns out to be very effective against a multitude of complex problems due to its powerful abstraction capabilities.

A given Alloy model is represented by a structure organized in:

ATOMS They represent the basic elements of the model. Each atom is characterized as *indivisible*, *immutable* and *uninterpreted* – it has no special properties on its own.

RELATIONS Each model's set of relations specify how its atoms interact with one another. A relation is defined by a set of tuples of atoms, where the order of atoms in a tuple matters – e.g. (a, b) and (b, a) represent two different tuples –, while the order of tuples inside the relation does not. Moreover, the properties associated with the model's atoms arise from the relations that they are a part of, meaning that various characteristics can be expressed, such as mutation, and other context specific attributes through them.

For every model there are three constant relations: `none`, the empty set; `univ`, the set containing all this model's atoms and `iden`, the identity relation, whose tuples are of the form (a, a) for every atom a in the model.

In order to reason about relations of the same arity, Alloy provides the set operators displayed in Table 2.3.

Union	$A + B$	$A \cup B = \{x \mid x \in A \vee x \in B\}$
Intersection	$A \& B$	$A \cap B = \{x \mid x \in A \wedge x \in B\}$
Difference	$A - B$	$A - B = \{x \mid x \in A \wedge x \notin B\}$
Subset	$A \text{ in } B$	$A \subseteq B \equiv \forall x. x \in A \Rightarrow x \in B$
Equality	$A = B$	$A = B \equiv \forall x. x \in A \Leftrightarrow x \in B$

Table 2.3: Set operators supported by Alloy.

Furthermore, Alloy implements relational operators like:

ARROW PRODUCT $A \rightarrow B$ is the relation containing all possible combinations of every tuple of each relation. If both relations are unary, their arrow product corresponds to set Cartesian product $A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$.

COMPOSITION $S \cdot R$ can be achieved through two possible operators: **Dot Join** $R.S$ or **Box Join** $S[R]$. While both operators are semantically equivalent, they vary in syntax and precedence (dot join has higher priority than box join). The coexistence of the two provide flexibility to the user, promoting the readability of the specification, allowing composition in a more *navigational* way with the first and in a *dereferencing* manner using the other.

If S, R tuples are of the form (s_1, \dots, s_n) and (r_1, r_2, \dots, r_m) respectively, then their composition is defined as $(s_1, \dots, r_2, \dots, r_m)$ for every tuple combination where s_n is equal to r_1 .

TRANSPOSE $\sim R$ represents the converse of a binary relation R .

TRANSITIVE CLOSURE \hat{R} is defined as the smallest *transitive* relation containing the binary relation R , i.e.,
 $\hat{R} = R + R.R + R.R.R + \dots$
 $*R$ represents the **Reflexive-Transitive Closure** of R , similarly to \hat{R} , while also being *reflexive*, that is, $*R = \hat{R} + \text{iden}$.

DOMAIN/RANGE RESTRICTIONS Given a relation R and a set S ,

$S <: R$ is the relation whose tuples are those of R , such that, their first atom is an element of S and

$R >: S$ contains all the tuples of R that have their last atom belonging to S .

OVERRIDE Similar to union, $R ++ S$ has every element of R and S , with the exception of tuples from R that have the same initial atom as another tuple in S – they are “replaced” by that tuple of S .

For example, $\{(a_1, b_1), (a_2, b_2)\} ++ \{(a_1, b_2), (a_3, b_1)\} = \{(a_1, b_2), (a_2, b_2), (a_3, b_1)\}$.

Constraints over relations must be specified using first-order logic formulas. In Table 2.4 are shown the logical operators implemented in Alloy. The alternative operator must be used with implication

and together they allow the specification of expressions of the kind `if P then A else B` written as `P implies A else B`.

		Alloy syntax	
Negation	\neg	not	!
Conjunction	\wedge	and	&&
Disjunction	\vee	or	
Implication	\Rightarrow	implies	=>
Alternative		else	,
Equivalence	\Leftrightarrow	iff	<=>

Table 2.4: Logical operators supported by Alloy.

Quantified properties are of the form `Q V | F`, where `Q` is the quantifier over the variables `V` – defined as `a1, a2, ... : A, b1, b2, ... : B, ..` – which appear on the formula `F`. `Q` can assume one of the values presented in Table 2.5.

\forall	all	F is valid for every <code>v</code> in <code>V</code>
\exists	some	F is valid for at least one <code>v</code> in <code>V</code>
$\exists!$	one	F is valid for exactly one <code>v</code> in <code>V</code>
	lone	F is valid at most for one <code>v</code> in <code>V</code>
	no	F does not hold for any <code>v</code> in <code>V</code>

Table 2.5: Alloy quantifiers.

Other than `all`, every other quantifier can be applied over an expression `e` as `Q e`, such that `some e` holds if `e` is not empty; `no e` holds if `e` is empty; `one e` is valid if `e` has exactly one tuple and `lone e` if `e` has, at most, one element.

Using the keyword `disj` in `V`, only instances where the variables are disjoint can be considered, e.g. `disj a, b : X` only examines instances where `a` and `b` have different values.

A relation can be defined by comprehension with

$$\{ x1 : S1, x2 : S2, \dots, xn : Sn \mid F \}$$

meaning its elements are of the shape `(x1, x2, ..., xn)` for the values where `F` holds.

The cardinality of a set is calculated using the operator `#`, which evaluates into an integer value. To reason about integers the following standard operations are provided: `+`, `-`, `=`, `<`, `>`, `<=` (less than or equal) and `>=` (greater than or equal).

At this point, we have seen how an Alloy model is structured, its components operations and how we can define properties over them. Thus, let us take a look at how we can precisely construct a model, impose desired constraints and so on, using the Alloy language.

A *signature* defines a set of atoms, and is declared with the keyword `sig`, e.g. `sig A{}` specifies a set A . Each signature can be further characterized through object-oriented alike concepts provided by the language:

- `sig B extends A{}` presents the set B as a subset of A and is called an *extension* of A . Note that two extensions of the same set are mutually disjoint. In case it is desired that the extensions may share atoms, `in` can be used in place of `extends`, e.g. `sig B,C in A{}`.
- `abstract sig A{}` contains no elements of its own but the atoms of its extensions.

Furthermore, the number of atoms of a given signature can be restricted by taking advantage of *multiplicity keywords*, presented in Table 2.6, which represent set multiplicity constraints. For instance, one `sig A{}`

set	Any number of elements
some	At least one
one	Exactly one element
lone	At most one

Table 2.6: Multiplicity keywords.

declares A as a singleton set.

To specify relations over these sets, one must declare them as *fields* inside the signature which corresponds to their domain, for example, `sig A{ R : B -> C }` defines the ternary relation $R : A \rightarrow B \rightarrow C$, whose tuples are of the form (a, b, c) with $a \in A$, $b \in B$, $c \in C$. Also, the multiplicity keywords introduced previously in Table 2.6 can be used inside the field declaration: `sig A { R : m B }` with m being one of the possible multiplicities, specifies how many elements of B can be associated with each atom of A or `sig A { R : B m -> n C }` constraints the number of occurrences of an element from C together with an element of B in a tuple of R according to multiplicity n and vice versa with multiplicity m .

Properties and requirements from the scenario that is being modeled can be encoded through the following possible special blocks:

FACTS Every constraint inside a `fact` section is assumed to always hold. Signature specific facts can be specified via *signature facts*, which implicitly quantifies over its atoms, by `sig A{ fields }{ facts }`.

ASSERTIONS The formulas inside the `assert` blocks express properties that are expected to arise from the model's facts. Afterwards they are to be verified by the Alloy Analyzer in order to check if that is the case, detecting potential inconsistencies in the specification otherwise.

PREDICATES Defines a reusable parametric constraint which, unlike facts, is not expected to always be valid, unless so specified.

```

pred name[arg1 : S1, arg2 : S2, ..., argN : SN] {
    constraint(arg1, arg2, ..., argN)
}

```

Using predicates, one can generalize formulas, improving the model readability and freely impose them only when desired. They are also adequate to model operations and mutation.

FUNCTIONS Being declared analogously to predicates, functions encapsulates an expression whose resulting relation must be specified, depending on zero or more arguments.

```

fun name[arg1 : S1, arg2 : S2, ..., argN : SN] : R1->R2->...->RN {
    expression(arg1, arg2, ..., argN)
}

```

Alloy also provides a polymorphic module system, allowing the division of one model in many modules, each corresponding to one *.als* – Alloy specification file extension – file. Each individual module can then be used in multiple different models.

A module can be defined with `module pathTo/moduleName[S1, S2, ..., SN]`, meaning it is parametric in zero or more signatures *S1, S2, ..., SN*. When importing a parametric module, all the signatures that it depends on must be specified, and thus, the same module can be instantiated more than once, for example:

```

open pathTo/example[X] as EX
open pathTo/example[Y] as EY

```

instantiates the module `example` with two different signatures *X* and *Y* and associates with each instance an alias *EX* and *EY*, respectively. Afterwards, their relations, predicates, functions and so on, are freely accessible and may be reasoned over.

Having completed an initial specification, the next steps include model analysis and gradual refinement. For that, some *commands* must be specified, which can be one of two types:

RUN Alloy Analyzer will attempt to find instances for which the predicate associated with the *run* command is valid according to the background specification – relations, facts, and all – in which it is being executed.

CHECK This command is applied to assertions, meaning the analyzer will try to identify a valid instance of the model for which the assertion does **not** hold.

Associated with every command there is a **scope** that constraints the number of atoms contained in each signature that will be considered in possible instances, i.e., it defines an upper limit to all the combinations

that can be examined for that model. Therefore, it turns out that the underlying logic is undecidable², meaning that if no instances are found for a *run* command, it does not necessarily mean that the model itself is inconsistent, just that for that specific scope there is not a single instance that satisfies all its constraints. Similarly, if no counterexamples are found for a *check* command, it does not mean that the assertion always holds, but that it is valid up to the scope considered only. However, if an instance is found for a *run* command, the user can be confident that the model is consistent and if the tool recognizes a counterexample for a given *check* command, then it can be established that the property does not hold in general. While this notion of scope may seem limiting at first, it allows Alloy to be a fully automated analyzer and reveals itself as very effective against many problems due to the relevance of the *small scope hypothesis* – “Most bugs have small counterexamples.” (Jackson, 2012).

Underneath the Alloy Analyzer there is a relational model finder, **Kodkod**. In order to execute a command, the analyzer translates the given Alloy specification into a Kodkod problem, which in turn is optimized by Kodkod and later represented in CNF so that it can be sent to one of the very efficient SAT Solvers. If an instance/counterexample is found, the resulting Kodkod instance is then interpreted by the Alloy Analyzer into a valid Alloy model.

Finally, given an Alloy model, it can be examined through the Alloy Analyzer GUI features:

VISUALIZER Portrays the instance in different representations – graph, text, table or tree. For the graphic presentation the modeler is able to create themes that customize its appearance, making it possible to come up with a more intuitive look for the model, easing the analysis.

EVALUATOR A terminal where the user can specify expressions in the Alloy language and evaluate their value with respect to the current instance.

2.7 SUMMARY

Through this chapter, the theoretical concepts required to accomplish the ambitions of this dissertation were established.

It started by highlighting essential *Relational Algebra* concepts, followed by their transition into the typed matrices universe. Afterwards, presented the distinction between the Boolean satisfiability problem and the satisfiability modulo theories problem and examined the corresponding SAT solving and SMT solving methodologies. Lastly, glanced over probabilistic models, costs/rewards, associated logics and their model checking techniques.

Finally, the Alloy language was introduced, alongside the Alloy Analyzer and its features.

² Note that the undecidability of the logic does not emerge from the notion of scope, but from the undecidability of the underlying first-order logic.

STATE OF THE ART

To progress with this project and be able to meet its aims, there is the need to research tools expressive enough to handle the theoretical background established in the previous chapter as well as relevant work acting on this field.

Thus, the next section presents and reviews the tools that will be used in the technical component of this dissertation, highlighting their capabilities and features.

Then, this chapter presents the study of previous work that addresses certain topics related to the focus of this project: on one hand, in Section 3.2.1 an approach taken to introduce the ability to reason about *multiconcepts* in Alloy is discussed; on the other hand, Section 3.2.2 inspects a way of using an SMT Solver as the backend to the Alloy Analyzer, instead of the conventional SAT Solvers.

3.1 TOOLS

3.1.1 *Satisfiability Modulo Theory Problem Solvers*

SMT solvers are automated theorem provers which can determine the satisfiability of a first-order logic formula according to a specific background theory. Such solvers have a large variety of applications, including software verification, test case generation, (bounded) model checking, planning and scheduling problems and others.

The specification which is fed into one of the various off-the-shelf solvers, for most of them, is written in a language conforming to the *SMT-LIB standard*. This norm aims to connect researchers on this topic in order to quicken its advancements, establishing a common ground between them by offering precise descriptions of background theories, managing a library of benchmarks for existing SMT solvers and making it available to everyone, while also providing a language to specify terms and formulas in SMT-LIB's underlying logic – many-sorted first-order logic with equality; define new background theories from already existing ones; specify the logical fragments whose formulas satisfiability will be verified according to a certain background theory and execute commands to reason about the specification, like assert or remove formulas, check their satisfiability and obtain the resulting model or unsatisfiability proof accordingly.

The SMT-LIB language can be characterized as follows:

SORTS Associated with every term is a *sort*, meaning it is typed and thus, represents a *well-sorted* term. A sort can be parametric, like `List A` varying with sort `A`, or not, which is the case of sort `Bool`.

TERMS A *term* can be built through *variables*, *function symbols*, *binders* and so on. A binder can either be a **Parameter Binder** (`par (s1 ... sn) e`) used to introduce sort parameters `s1`, ..., `sn` in an expression `e`, or a **Variable Binder**, which introduces one or more variables locally:

- `forall` and `exists` binders specify the universal and existential quantifiers, respectively. For example,

```
(exists ((x1 s1) (x2 s2)) e)
```

quantifies the expression `e` over the variables `x1` of sort `s1` and `x2` of sort `s2`.

- `let` declares multiple local variables in parallel, e.g.

```
(let ((x1 t1) (x2 t2)) t)
```

describes `t` with all the free occurrences of `x1` and `x2` replaced by the terms `t1` and `t2` accordingly.

- Pattern matching of algebraic data types can be achieved with `match`, for example,

```
(match l (
  (nil true)
  (_ false)))
```

matches the list `l` to `true` if it is empty and to `false` otherwise.

THEORY DECLARATION Theory `T` is described through a declaration of the form

```
(theory T (a1 ... an)),
```

where `a1`, ..., `an` represent its *attributes*, in particular:

- `:sorts` are specified the sort symbols included in the theory;
- `:funs` contains the declarations of the function symbols considered.

LOGIC DECLARATION A fragment of the SMT-LIB main logic can be specified in the form `(logic L (a1 ... an))`, with `a1`, ..., `an` being the attributes of the logic `L`, for instance, `:theories` includes the list of theory names from the standard to be considered in `L`.

SCRIPTS Sequences of *commands*. Their purpose is to define the interaction with the SMT solver such that, it reads one command at a time, acts accordingly, produces a response and then repeats the process on the following command until it hits one termination condition.

In general, solvers that abide to the standard use an **Assertion Stack** to manage the scripts. This stack has *levels* for elements, which consist of sets of *assertions*. Each assertion can be a logical formula – a term of sort `Bool` – or sort/function symbols specifications. *context* refers to the union of all assertion levels in the stack alongside global declarations. New levels can be introduced with the command `push` or removed using `pop`, meaning that the assertions introduced in those levels are discarded and ignored through the remainder of the commands. `reset` restarts the state of the solver to the moment before starting to read commands, emptying the assertion stack.

- `(set-logic L)` informs the solver that it should analyse the specification using logic *L*.
- `(set-option o v)` sets the option *o* to the value *v*, if supported by the solver. In particular, if set to *true*:
 - `:produce-models` enables the commands `(get-value (t1 ... tn))` – which returns the value of the specified terms *t1*, ..., *tn* in the current model – and `(get-model)` – provides the respective interpretation for every user-defined function symbol in the current model.
 - `:produce-proofs` enables `get-proof` which returns a proof of unsatisfiability for all the formulas considered in the current context.
 - `:produce-unsat-cores` enables the command `get-unsat-core` that requests an *unsatisfiable core* – a subset of all the formulas containing those that are unsatisfiable by themselves – for the current context.
- Sorts can be introduced with the commands `declare-sort` or `define-sort` for parametric sorts.
- `(declare-fun f (s1 ... sn) s)` is used to define a function symbol *f* of sort *s*, depending on sorts *s1*, ..., *sn*.
`(declare-const f s)` is equivalent to `(declare-fun f () s)`.
- `(assert t)` introduces the assertion *t* in the current stack level.
- `(check-sat)` causes the solver to attempt to find a model that satisfies all the assertions in the current level, resulting in one of three possible responses:
 - `sat` if a model was successfully found;
 - `unsat` if such model does not exist;
 - `unknown` if the search was inconclusive, which can be caused by reaching the *timeout* value for example.

If the response `sat` or `unknown` was obtained, `get-model` or `get-value` can be used; otherwise, after an `unsat` response, commands like `get-unsat-core` or `get-proof` may be used for further analysis.

An SMT-LIB logic may consider one or more of the following theories:

ArraysEx – Functional arrays with extensionality, equipped with read and write operations `store` and `select`, respectively.

FixedSizeBitVectors – Theory of arbitrary length BitVectors with:

- `concat` and `extract` for bitvector concatenation and extraction, respectively;
- Logical operators `bvop`, where `op` represents one of `not`, `and`, `or` or `xor`;
- Arithmetic operators `bvop` with `op` being `neg`, `add`, `sub`, `mul`, `div` or `rem`;
- `bvshl` for left bit shifting and `bvlsbr` for unsigned logical right shifting;
- ...

Core – Describes the basic Boolean operators `not`, `=>`, `and`, `or` and `xor`, along with `=` for Boolean equality, `distinct`, which returns `true` if and only if its arguments are not identical, and `ite` used to represent an `if..then..else` - like expression.

FloatingPoint – A theory for floating point numbers.

Reals – The theory of real numbers which describes `-` (negation or subtraction depending on the number of arguments) alongside the rest of arithmetic operators `+`, `*`, `/` and the comparison functions `=`, `<`, `>`, `<=` and `>=`.

Ints – Integer Arithmetic theory which implements the integer version of the operations described before for the theory of reals, with `div` for integer division instead of `/` together with the remainder of the Euclidean division `mod` and the absolute value operation `abs`.

Reals_Ints – A theory that supports both integer and real numbers together, analogously as previously described for their individual theories.

Figure 3.1 showcases the different sub-logics from the SMT-LIB base logic included in the standard. Each fragment's designation follows a naming convention, for instance, **QF** stands for *quantifier-free formulas*, **IA** identifies the theory of integers and the rest defined likewise.

In order to solve an SMT problem, the solver usually implements one of two main approaches:

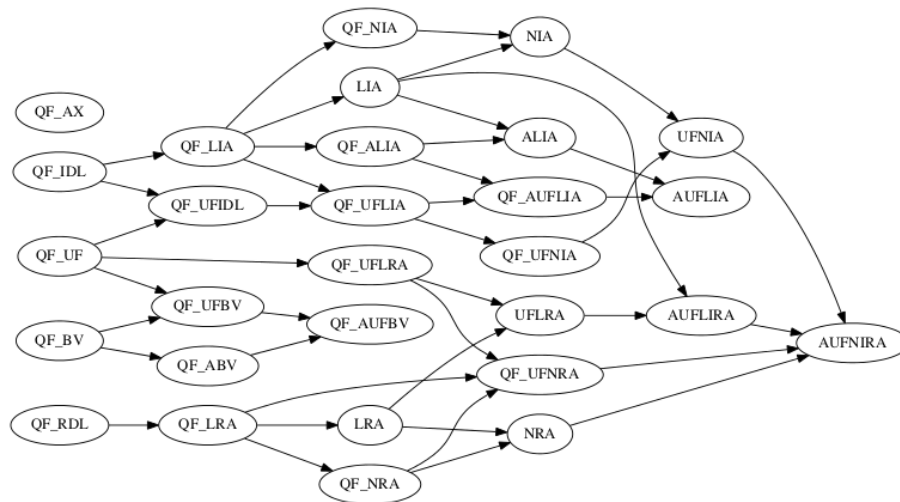


Figure 3.1: SMT-LIB logic fragments (Barrett et al., 2016).

“EAGER” The “eager” approach involves translating the initial problem into an equisatisfiable propositional formula, to be solved by a SAT solver afterwards. Thus, this methodology needs to use all the theory information from the very beginning, requiring sophisticated encodings for each of them.

STP (Ganesh and Dill, 2007), Beaver (Jha et al., 2009) and Boolector (Niemetz et al., 2014) are some examples of solvers that follow this approach.

“LAZY” The “lazy” methodology takes advantage of *Boolean abstraction* in order to obtain an overapproximate SAT formula from the initial SMT formula, to be subsequently passed into a DPLL based SAT solver. A theory decision procedure is used to continuously refine the abstraction and manage the SAT solver.

Consequently, this approach only uses the theory information as needed through the process.

Due to the nature of the procedure – the theory and SAT solvers communicate with each other via a simple API –, a new theory simply requires an adequate solver for that theory and any off-the-shelf (DPLL-based) SAT solver can be easily integrated into a “lazy” SMT solver, resulting in a modular and flexible approach.

SMT solvers that follow this methodology include CVC4 (Barrett et al., 2011), Z3 (de Moura and Bjørner, 2008) and Yices (Dutertre, 2014).

3.1.2 PRISM

PRISM (Kwiatkowska et al., 2011) is an open-source probabilistic symbolic model checker, which allows the development of probabilistic models of many kinds, supporting *Discrete-Time* and *Continuous-Time*

Markov Chains, Markov Decision Processes and *Probabilistic (Timed) Automatas* also. Moreover, qualitative and quantitative properties can be specified over a given PRISM model, to be automatically analysed afterwards. Its property specification language includes temporal logics like PCTL and LTL, while also supporting quantitative specification and reasoning about costs and rewards. Therefore, this tool is ideal to handle problems that are inherently probabilistic or non-deterministic.

In order to model a problem at hand, one takes advantage of the offered *state-based* language:

MODEL TYPE The PRISM specification must state type of the probabilistic model that is being considered.

One model can have one of the types mentioned previously, indicated using the keyword `dtmc`, `ctmc`, `mdp` or `pta`.

MODULES Inside one model may exist multiple processes, each described by a **module**.

```
module P
    ...
endmodule
```

A module is described by its **variables**, which define its state, and **commands**, that specify how its state evolves over time.

VARIABLES In general, a variable can be an integer or a Boolean, and can be declared as follows:

```
a : [1..5];
b : bool init true;
c : int init 10;
```

This example shows `a` as a bounded integer assuming values between 1 and 5, `b` is a Boolean variable with value *true* in the initial states and `c` represents an unbounded integer with initial value 10.

If declared inside a module, a variable is local to it. However, one can declare a global variable outside module declarations with the keyword `global`, being accessible by any process in that case.

COMMANDS A command is of the form `[a] g -> us`; where `a` is an *optional* action label, `g` defines a *guard*, i.e., a Boolean expression, and `us` specifies one or more *updates*.

A single update is of the form `(x' = e)`, setting the value of the variable `x` to the expression `e` – which must evaluate to the same type of `x`. There are various ways to specify the updates on a single command, for example, multiple updates can be combined through `&`, different updates for the same variable depending on probabilities can be modeled through probabilistic choice with `+`, along with other operators.

```
[ ] a < 5 -> 0.30 : (a' = a) + 0.7 : (a' = a + 1);
[ ] b -> (b' = false)&(c' = 10 + a);
```

In this example, the first command states that if the current value of a is less than 5, a is incremented by 1 with 70% probability or remains unchanged with probability 30%; the second transition sets the value of b to *false* and the value of c to the current value of a plus 10, in case b is *true*.

Local non-determinism, for the model types that support it (e.g. MDP), can be achieved through multiple commands with the same guard, for instance:

```
[ ] a > 3 -> (b' = !b);
[ ] a > 3 -> (b' = b);
```

if a 's value is higher than 3, it may flip b 's value or leave it unchanged.

The use of an action label allows for process *synchronization*, meaning that state transitions with the same label between modules must occur at the same time.

CONSTANTS Constants of type integer, double or Boolean can be used.

```
const int x;
const bool xIsPositive = x > 0;
const double y = xIsPositive ? 0.4 : 2.5;
```

In this case, x is an integer constant without a value assigned, $xIsPositive$ is a Boolean constant indicating if x is positive or negative and the double y has value 0.4 if x is positive and value 2.5 otherwise.

EXPRESSIONS As seen throughout the presented examples, *expressions* take part in multiple concepts of the language. An expression can include combinations of concrete values, variables, constants, operators, and so on.

PRISM provides the operators and functions highlighted in Table 3.1.

Numeric		Logic		Functions	
Minus/Subtraction	-	Equality	=	Minimum value	min(...)
Addition	+	Not equal	!=	Maximum value	max(...)
Multiplication	*	Negation	!	Floor	floor(v)
FP Division	/	Conjunction	&	Ceiling	ceil(v)
Less than	<	Disjunction		Round	round(v)
LT or Equal	<=	Equivalence	<=>	Power b^e	pow(b, e)
Greater than	>	Implication	=>	Integer modulo	mod(a, n)
GT or Equal	>=	Alternative	· ? · : ·	Logarithm $\log_b x$	log(x, b)

Table 3.1: Expressions operators and functions supported by PRISM.

X	Next
U	Until
F	Eventually
G	Always
W	Weak Until
R	Release

Table 3.2: Temporal operators.

FORMULAS A *formula* associates a name to an expression, being useful to avoid duplicate code. Each formula is declared as `formula n = e;`, where *n* is the name that will represent the expression *e*. A formula can be used anywhere in model any number of times.

INITIAL STATES If one model can have multiple initial states, they can be defined inside a `init .. endinit` block, instead of using `init` in variable declarations inside modules, like seen previously.

COSTS AND REWARDS States/Transitions of the model may or may not have costs/rewards associated through `rewards ... endrewards` sections.

In order to assign rewards/costs to states, each *reward item* inside the block must be of the form `g : r;`, where *g* is a guard over *all* the variables in the model and *r* is an expression describing the reward for that particular state.

In case the user intends to associate rewards/costs to state transitions, the reward items have to be of the form `[a] g : r;` where *a* is an action label, meaning that the reward *r* will be considered for the transitions with that label that satisfy the guard *g*.

PRISM's property specification language allows one to reason about different kinds of properties. A *path property* is a Boolean-valued formula over a specific path of the model. Due to the nature of the logics included in the language, in a path property can occur the temporal operators presented in Table 3.2.

The P operator is used on properties that deal with the probability of some occurrence, being of the form $P\Box p$ [*path property*], where $\Box \in \{>, <, \geq, \leq\}$ and *p* represents a double value between 0 and 1, evaluating to *true* or *false*, or of the form $P=?$ [*path property*], which measures the probability in question. All the temporal operators presented before have a *bounded* version (with the exception of X), for instance $P<0.5$ [$F\leq 3$ $a=2$] indicates “a is equal to 2 within 3 time units with less than 50% probability”.

In order to reason about the behaviour of the model in the *long-run* one can use the S operator, specifying properties in a similar manner to the P operator. However, unlike the previous, the S operator only supports Boolean properties (instead of path properties). For example, $S\geq 0.5$ [$x > 3$] checks if the probability of *x* being greater than 3 in the long-run is greater or equal to 50%. Alternatively, properties like

$S=? [y + z < 10]$ measure the steady-state probability of the Boolean expression at hand, in this case, the addition of y with z never reaching 10.

Reward-based properties are specified with the R operator, whose form is analogous to that of the P operator, while also adding operators C and I to deal with cumulative and instantaneous rewards, respectively. PRISM takes into account the *expected value* of the rewards/costs when dealing with this type of properties. A reward property may be used to examine:

- Reachability reward: $R \square \text{reward} [F \text{ property}]$;
- Cumulative reward: $R=? [C \leq \text{reward}]$;
- Total reward: $R=? [C]$;
- Instantaneous reward: $R \square \text{reward} [I = \text{instant}]$;
- Steady-state reward: $R \square \text{reward} [S]$;
- ...

Moreover, non-probabilistic properties can also be verified with PRISM, if specified using CTL or LTL. These properties are of the form $Q [\text{path property}]$, where Q is one of the two *path quantifiers* – A and E, the universal and existential quantifiers over paths, respectively.

After building a PRISM model, the tool yields access to a number of features to analyse it, in particular:

SIMULATOR By taking advantage of the simulator, PRISM generates a valid path of a certain length – in terms of state transitions or time taken, depending on the type of the model considered – for the current model. If the model has costs/rewards, the user can check the value for each state and the accumulated value throughout the path.

MODEL CHECKING Property verification of the formulas specified can be performed through PRISM. If a property is qualitative, the result comes as *true* or *false*; for quantitative properties the tool displays the resulting numeric value instead.

STATISTICAL MODEL CHECKING For a subset of the kinds of properties supported by PRISM, one can generate approximate results for them, according to one of the methods implemented by the tool to perform *statistical model checking*:

- Confidence Interval;
- Asymptotic Confidence Interval;
- Approximate Probabilistic Model Checking;
- Sequential Probability Ratio Test.

This type of model checking is most useful when dealing with large models, where the usual model checking procedure is infeasible.

EXPERIMENTS When the model contains one or more undefined constants, PRISM requires the user to specify their value before being able to verify a given property. *experiments* provide a way of defining a range of values for each constant, and then will check the property for each possible combination. After the experiment is complete, the results can be exported and/or analysed through the tool's GUI, which is able to plot the corresponding graph.

PARAMETRIC MODEL CHECKING Instead of experiments, for a specific subset of valid properties, one can use *parametric model checking*, which is able to determine a rational function over the parametric constants or mappings from a range of values for these parameters to rational functions or Boolean values.

3.2 RELATED WORK

3.2.1 Encoding Multirelations in Alloy

Instead of assuming the types of relations as sets, they can be relaxed into *bags* (or *multisets*), which allow multiple copies of the same element. Furthermore, by considering, for example, a binary relation $R : A \rightarrow B$ not as a set whose elements are pairs (a, b) with $a \in A$ and $b \in B$, but as a *bag*, then it forms a *multirelation*, meaning that the same pair (a, b) can occur multiple times in the relation R . As can be seen, the notion of *multirelation* allows for quantitative property modelling, the theme at target in this dissertation.

Sun et al. (2016) provide a way of introducing these concepts in Alloy using an approach which they call *index-based*. This way of reasoning about *multirelations* has its basis in *Category Theory*, namely by relying on the concepts of **spans** and **pullbacks**.

To achieve finer control when reasoning over relations in categorial fashion, closer to the way of handling them pointwise while still maintaining a certain degree of generality, one can work in the narrowed down context of **tabular allegories** (Bird and de Moor, 1997).

Additionally to the conditions that identify a proper category, an **allegory** \mathbb{A} is enhanced with three operators:

INCLUSION Any two morphisms $R, S : A \rightarrow B$ of \mathbb{A} can be compared under a partial order \subseteq , as $R \subseteq S$ for instance, as long as the order ensures monotonicity of composition, that is:

$$(R_1 \subseteq R_2) \wedge (S_1 \subseteq S_2) \Rightarrow R_1 \cdot S_1 \subseteq R_2 \cdot S_2$$

for every pair of comparable arrows R_1, R_2 and S_1, S_2 , and every composable pair of morphisms R_1, S_1 and R_2, S_2 in \mathbb{A} .

MEET Given arrows $R, S : A \rightarrow B$, there must be another arrow $R \cap S : A \rightarrow B$ present in \mathbb{A} conforming to the following universal property:

$$X \subseteq (R \cap S) \equiv (X \subseteq R) \wedge (X \subseteq S)$$

for every $X : A \rightarrow B$.

CONVERSE Every arrow $R : A \rightarrow B$ can be “flipped” to obtain another arrow $R^\circ : B \rightarrow A$ meeting the following properties:

- **Involution** $(R^\circ)^\circ = R$
- **Order preservation** $R \subseteq S \equiv R^\circ \subseteq S^\circ$
- **Contravariance** $(R \cdot S)^\circ = S^\circ \cdot R^\circ$

A pair of functions $\langle f, g \rangle$, with $f : C \rightarrow A$ and $g : C \rightarrow B$, represents a **tabulation** of a given morphism $R : A \rightarrow B$ if the following conditions are met:

- (a) $R = g \cdot f^\circ$
- (b) The pair is *jointly injective*:

$$\begin{aligned} (f^\circ \cdot f) \cap (g^\circ \cdot g) &= id \\ \Downarrow \\ h = k &\equiv (f \cdot h = f \cdot k) \wedge (g \cdot h = g \cdot k) \end{aligned}$$

for all functions h, k .

When for each and every arrow contained in an allegory there exists, at least, one tabulation, the allegory is said to be **tabular**. *Rel* is an example of such tabular allegory: it is an allegory since it also supports all three operators, already described in Section 2.1, abiding to the properties required above; given any relation $R : A \rightarrow B$ from *Rel*, by choosing a suitable $C \subseteq A \times B$, then the pair $\langle fst, snd \rangle$ composed by the relational product projections corresponds to a tabulation for such R , in particular, $R = snd \cdot fst^\circ$, with the injectivity property holding under these conditions, as follows:

$$\begin{aligned} &(fst^\circ \cdot fst) \cap (snd^\circ \cdot snd) \\ &\equiv \{ \text{kernel of } fst \text{ and } snd \} \\ &\ker fst \cap \ker snd \end{aligned}$$

$$\begin{aligned}
 &\equiv \{ \ker \langle R, S \rangle = \ker R \cap \ker S \text{ (2.14)} \} \\
 &\quad \ker \langle fst, snd \rangle \\
 &\equiv \{ \times\text{-reflection (2.12)} \} \\
 &\quad \ker id \\
 &\equiv \{ \text{kernel of } id; \text{ identity (2.2)} \} \\
 &\quad id
 \end{aligned}$$

thus, Rel is also tabular.

A **span** $R : A \rightarrow B$ is represented by a triple (h_R, s_R, t_R) where h_R is a set defining its *head* and $s_R : A \leftarrow h_R, t_R : h_R \rightarrow B$ are functions which specify its *legs*. Then, relations can be perceived as spans, as illustrated in Figure 3.2: on the left, $R : A \rightarrow B$ represents an ordinary relation, defined by $R = \{(a_1, b_1), (a_1, b_3), (a_2, b_4), (a_3, b_2), (a_4, b_4)\}$, while on the right it shows the corresponding span (h_R, s_R, t_R) , where the *head* is indexing each original pair, and its *legs* are defined accordingly.

Having presented the definition of a tabulation previously, one can quickly check that the notion of span arises by relaxing the first, precisely by taking $C = h_R$ and $\langle s_R, t_R \rangle$ as the pair of functions, however without expecting the pair to be jointly monic. The structure of a span, besides representing ordinary relations, is also capable of describing *multirelations*, as illustrated in Figure 3.3, something that cannot be achieved with tabulations, as revoking the need for the pair to be mutually injective is what makes such representation possible, like will be exemplified soon.

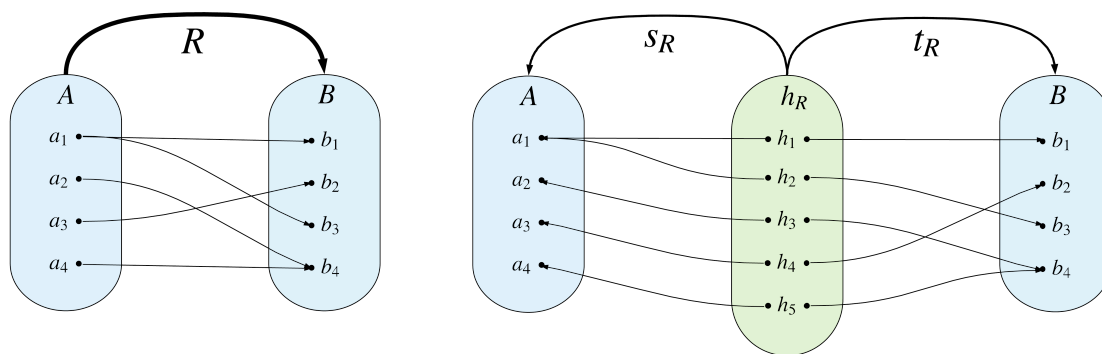
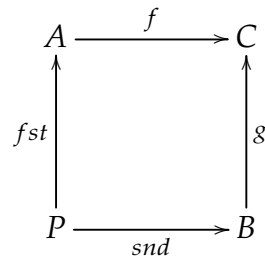


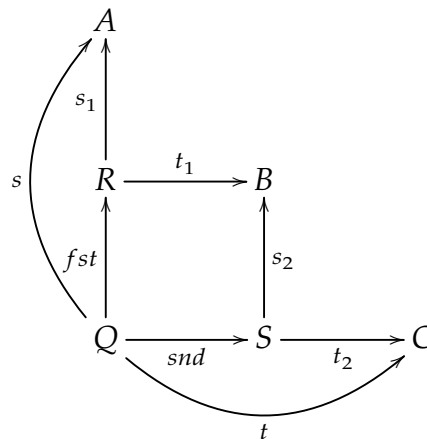
Figure 3.2: Example of a relation on the left, and its *multirelation* representation on the right.

The **pullback** of two functions $f : A \rightarrow C, g : B \rightarrow C$ is a span (P, fst, snd) , such that, its head is defined by $P = \{(a, b) \in A \times B \mid fa = gb\}$, that is, the following diagram commutes



i.e., $f \cdot fst = g \cdot snd$; and its *legs* coincide with the projections fst and snd , as expected by the relationship between spans and tabulations as well as the tabulation highlighted for the relations of the allegory Rel .

Now that the necessary categorial concepts were introduced, the composition of two *multirelations* can be properly specified by taking it as **span composition**, calling it *multijoin*. Given two composable *multirelations* $R : A \leftrightarrow B$ and $S : B \leftrightarrow C$, $S \cdot R = Q : A \leftrightarrow C$ serves as the produced *multirelation*, which is obtained as follows:



1. Compute the pullback between t_1 and s_2 to obtain the *head* of Q , i.e., to determine all the composable links between R and S ;
2. Find the *legs* by composition $s = s_1 \cdot fst$ and $t = t_2 \cdot snd$.

then, the resulting span (Q, s, t) corresponds to the expected *multirelation*. Figure 3.3 shows an example of such composition. Note that the chosen *multirelations* which are being composed correspond to ordinary relations, while the result of the composition is, in fact, a *multirelation*. Therefore, unlike Alloy, which assumes and guarantees that relational composition is closed under ordinary relations, in general, that is not the case, the composition of two ordinary relations can be a *multirelation*, which means that perceiving the result of composition as a relation results in loss of information, as discussed already by the authors.

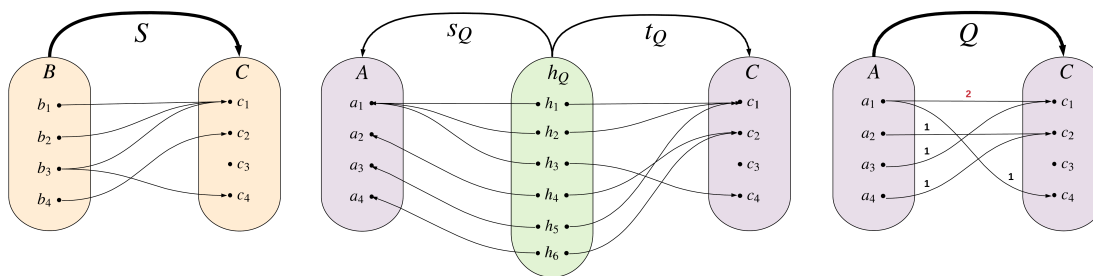


Figure 3.3: Q as the composition between S and R from Figure 3.2, portrayed as a span in the middle and as a relation whose arcs are weighted in the right.

Again, the representation of Q in Figure 3.3 is only possible since the structure allows for pairs of functions which are not injective, in particular by taking $h = \underline{h_1}$ and $k = \underline{h_2}$ with $f = s_Q$ and $g = t_Q$ in the previous definition of mutual injectivity:

$$\begin{aligned}
 & (s_Q \cdot \underline{h_1} = s_Q \cdot \underline{h_2}) \wedge (t_Q \cdot \underline{h_1} = t_Q \cdot \underline{h_2}) \\
 \equiv & \{ \text{going pointwise; composition; } \underline{c} a = c \} \\
 & (s_Q(h_1) = s_Q(h_2)) \wedge (t_Q(h_1) = t_Q(h_2)) \\
 \equiv & \{ \text{evaluate } s_Q \text{ and } t_Q \text{ with respect to Figure 3.3} \} \\
 & a_1 = a_1 \wedge c_1 = c_1 \\
 \equiv & \{ \text{Mutual injectivity} \} \\
 & \underline{h_1} = \underline{h_2} \\
 \equiv & \{ \text{Two constant functions are the same if their output constant coincides; } h_1 \neq h_2 \} \\
 & \perp
 \end{aligned}$$

one can establish that the legs of Q do not form a tabulation. However, the pair $\langle s_R, t_R \rangle$ from the span R depicted in Figure 3.2 does, as can be easily observed, both functions s_R and t_R are injective, meaning that the span is also a tabulation for the relation R . Thus, one can conclude that if a pullback meets all the conditions to also be considered a tabulation of a given relation R , then the *multirelation* is, in fact, an ordinary relation, since the repetition of arcs supported by *multirelations* is exactly the cause for the legs to not be injections. In the end, when working under *Rel* one can take advantage of tabulations to handle relations, while spans are adequate when handling *multirelations* from *MRel* instead, with the latter representing the category of binary *multirelations* with sets as objects and *multirelations* as arrows.

In order to go from relation to *multirelation* or vice versa, there are two operations which can be taken advantage of: **lift** – constructs a *multirelation* with the same information contained in the relation specified – and **drop** – given a *multirelation*, discards repeated arcs between the same two elements.

The implementation of these concepts in the Alloy language was achieved with the utilization of the Alloy (parametric) module system and thus, the library built is organized in the following modules:

`multi` – encapsulates useful operators on *multiconcepts* like `pullback` and `drop`;

`mrel[s, t]` – represents a *multirelation* $R : s \rightarrow t$ and provides the predicates `composedFrom`, which is true if the *multirelation* considered is the result of the composition of the two spans specified, and `liftedFrom` that allows the representation of this *multirelation* from an ordinary relation;

`mset[g]` – describes a *multiset* and implements operations over it.

Let us see a simple example of the usage of this library, the specification presented in Figure 3.4 and a possible instance displayed in Figure 3.5.

```

R : A ↔ B  open mrel[A, B] as R
S : B ↔ C  open mrel[B, C] as S
Q : A ↔ C  open mrel[A, C] as Q

R : A → B  sig A { r : set B }
S : B → C  sig B { s : set C }
             sig C {}

fact{
    R/liftedFrom[r]
    S/liftedFrom[s]
    Q/composedFrom[R/get, S/get]
    Q = S · R
}
    
```

Figure 3.4: Example of an Alloy specification dealing with *multiconcepts*.

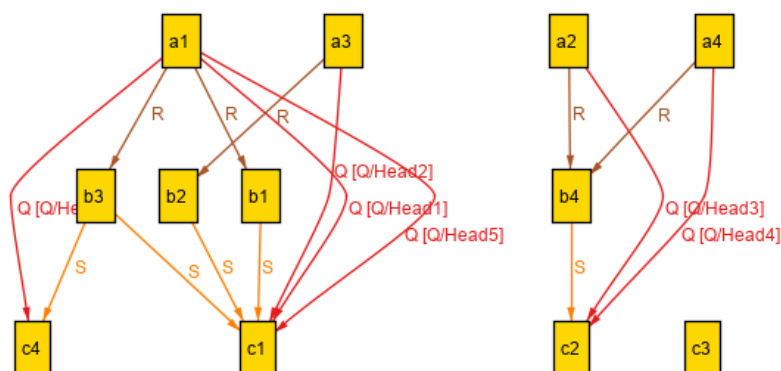


Figure 3.5: One valid instance for the Alloy model in Figure 3.4.

After experimenting with the library on a few examples and, in particular, extending already existing models to support *multiconcepts*, we come to the conclusion that the library is fairly easy to use and integrate, flexible and convenient for most small cases. However, this implementation also comes with some inconveniences:

- It is not scalable;
 - The more the user increases the scope, the harder it is on the Alloy Analyzer (even more so than usual);
 - The number of uses of *multiconcepts* in a model heavily impact the number of constraints that the solver has to deal with. Each *multirelation* declaration, composition, etc., accumulates to, at a some point, an unbearable amount of constraints associated with the specification of each *multiconcept*.
- The model can become very verbose and complex. Depending on the properties that are being specified using *multirelations*, the modeler needs to instantiate a new *multirelation* for each intermediate step which require *multiconcepts* – multijoin, disjoint union and others. For example, in order to compose two, already existing, ordinary relations, first they have to be lifted into into the *multirelation* realm, which means instantiating the module for each of them, and *then* the user still needs to instantiate the `mre1` module for the resulting *multirelation* which will hold the result of the composition. This process can easily get out of hand, and make the model very wordy and harder to read, analyse, debug and so on;
- Explicit measuring of quantitative information requires dealing with set cardinality and, by consequence, integer atoms, which should be avoided due to their implementation in Alloy and the limitations that arise when attempting to reason about them;
- Different *multirelations* with the same type cannot be easily declared due to the nature of the Alloy module system. However, this can be fixed by creating a preprocessor which makes copies of the library modules, but with different names, according to the model at hand;
- It does not explicitly support relations with arity higher than two.

In the end, it still proved itself as a very good solution for a multitude of problems of this kind and it was a great starting point in this investigation to find a more effective and practical way of dealing with quantitative properties in Alloy.

3.2.2 Relational Algebra and SMT Solvers

Due to the nature of quantitative properties that the envisioned Alloy extension will be able to deal with, SAT solvers lack the expressiveness required to effectively deal with them, which means that SMT solvers will be taken advantage of instead and thus, giving origin to the challenge of how to reason relationally using an SMT solver, in particular, how to connect Alloy to an SMT solver.

Yet, Meng et al. (2017) presented an Alloy extension to support the SMT Solver **CVC4**, which is being actively developed by the authors at the time of writing. The authors managed to construct a calculus which allows the translation and reasoning between Alloy syntax and SMT-LIB syntax.

In order to deal with relations, they developed a theory T_R (see Figure 3.6) as an extension of an already existing theory T_S for finite sets, where a sound, complete and terminating calculus exists, and is implemented in CVC4. In terms of correctness, the calculus designed for the relational theory T_R guarantees model and refutation soundness overall and termination for a specific fragment of the constraints language.

Set symbols:

$$\begin{aligned} [] : \text{Set}(\alpha) & \quad \sqcup, \sqcap, \setminus : \text{Set}(\alpha) \times \text{Set}(\alpha) \rightarrow \text{Set}(\alpha) & \in : \alpha \times \text{Set}(\alpha) \rightarrow \text{Bool} \\ [-] : \alpha \rightarrow \text{Set}(\alpha) & \quad \sqsubseteq : \text{Set}(\alpha) \times \text{Set}(\alpha) \rightarrow \text{Bool} \end{aligned}$$

Relation symbols:

$$\begin{aligned} \langle -, \dots, - \rangle : \alpha_1 \times \dots \times \alpha_n & \rightarrow \text{Tup}_n(\alpha_1, \dots, \alpha_n) \\ * : \text{Rel}_m(\alpha) \times \text{Rel}_n(\beta) & \rightarrow \text{Rel}_{m+n}(\alpha, \beta) \\ \bowtie : \text{Rel}_{p+1}(\alpha, \gamma) \times \text{Rel}_{q+1}(\gamma, \beta) & \rightarrow \text{Rel}_{p+q}(\alpha, \beta) \text{ with } p+q > 0 \\ -^{-1} : \text{Rel}_m(\alpha_1, \dots, \alpha_m) & \rightarrow \text{Rel}_m(\alpha_m, \dots, \alpha_1) & + : \text{Rel}_2(\alpha, \alpha) \rightarrow \text{Rel}_2(\alpha, \alpha) \end{aligned}$$

where $m, n > 0$, $\alpha = (\alpha_1, \dots, \alpha_m)$, $\beta = (\beta_1, \dots, \beta_n)$, and $\text{Rel}_n(\gamma) = \text{Set}(\text{Tup}_n(\gamma))$.

Figure 3.6: Signature Σ_R of the theory T_R (Meng et al., 2017).

When verifying properties in Alloy, the analyzer can prove (find counterexamples) that a property does not hold for a given model, but it can never prove that it *always* holds, even if no counterexample was found, only that it is valid up to the specific scope which was considered by the user. The extension aims to overcome this obstacle, making the tool more powerful so that, when possible, the modeler is able to check the validity of properties of Alloy models, having confidence that they are valid in case no counterexamples are found.

Moreover, this extension also provides a more effective way of dealing with the Alloy integers, adding support for constraints over unbounded integers, which means that they are no longer bounded by a given bit width.

Beyond the subject of verification, due to the CVC4 model finding capacities, it is able to determine minimal satisfying interpretations for consistent Alloy specifications or minimal counterexamples of specified properties without the modeler having to provide a scope, while still supporting scope specification in case the user so desires.

Although in some cases the solver might not be able to determine the satisfiability of certain formulas, due to the undecidability of the theory considered, there are already multiple instances where the designed calculus is able to terminate. The active development of this extension and research for decidable fragments also promotes the evolution and increase in power of this tool, making it a potential valuable resource in this project.

3.3 SUMMARY

At this point, the focused tools together with relevant state-of-the-art work on this subject were analysed.

This chapter began by identifying the key tools – SMT Solvers and PRISM – needed to accomplish the desired Alloy extension and inspecting their capabilities.

In the end, presented the introduction, discussion and evaluation of existing approaches similar to the goals of this dissertation, one way of encoding quantities in Alloy specifications through *multirelations* and the integration of a specific SMT solver in the Alloy Analyzer to overcome the current limitations of the implemented SAT solving approach.

THE PROBLEM AND ITS CHALLENGES

As stated in the introduction, the main aim of this project is to design an extension to the Alloy tool able to cope with quantitative models in problem specification. This chapter starts by proposing three case studies that will be used to benchmark such an extension to Alloy. In the next sections, some case studies are addressed using off-of-the-shelf tools, while others are modeled quantitatively by taking advantage of the theoretical foundations introduced before. The idea is to compare the outcome of this project, presented in Chapter 6, against the performance of such standard solutions. In the end, an initial design for the Alloy quantitative extension is proposed, as well as the motivation behind such architecture.

4.1 CASE STUDIES

This section presents some of the case studies to which the developed solution will be applied, and compared against using state-of-the-art tools.

Section 4.1.1 introduces a quantitative bibliographic system and tackles it using typed linear algebra in $Mat_{\mathbb{N}_0}$.

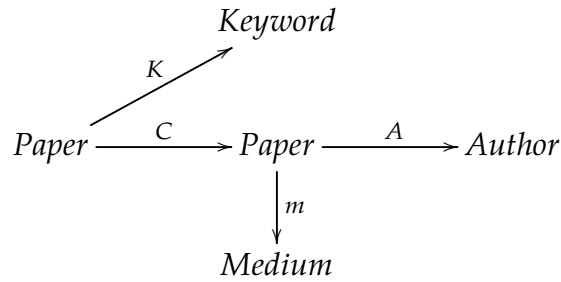
A football championship scheduler is modeled using Alloy in Section 4.1.2 and is further quantitatively analysed by taking advantage of an SMT Solver.

Both the *Bibliometrics* and *Football Championship* case studies are expected to be handled by the quantitative extension of Alloy through quantitative analysis under $Mat_{\mathbb{N}_0}$ and, at this stage, they are being modeled through two different means simply for illustration purposes, as a starting point. Going over them the other way around, that is, using SMT Solvers to study the bibliographic database scenario and modeling the football tournament through typed linear algebra would also be possible.

Finally, 4.1.3 presents a probabilistic model (Markov chain) of the behaviour of a grass sprinkler depending on whether it is raining or not. Initially, the problem is modeled using typed linear algebra in LS , and then encoded and model checked using PRISM.

4.1.1 Bibliometrics

A standard relational model of a bibliographic system managing papers, authors and citations across different scientific research fields could be as follows, taking advantage of the abstract diagram notation in *Rel* (Oliveira, 2017b),



where the relations depicted specify the following:

- $q C p$ means that paper p cites paper q ;
- $a A p$ specifies that a is one of the authors of paper p ;
- $k K p$ indicates that paper p mentions the keyword k ;
- $u = m p$ indicates u as the publication medium of paper p .

Since a paper cannot cite itself, the constraint $C \cap id \subseteq \perp$ is required. Such model can be encoded using Alloy where, for instance, the previous constraint is denoted by the Alloy fact block `fact{ no C & iden }`. The full specification of this (toy) bibliographic system can be found in Appendix A.1.1.

Augmenting the given model by introducing the relation $Keyword \xleftarrow{S} Paper = K \cap K \cdot C^\circ$ allows for the characterization of papers that are cited in the same area:

$$k S p \Leftrightarrow k K p \wedge \langle \exists q : p C q : k K q \rangle$$

In words: “paper p is cited by *at least* another paper q within the same field k .”

Furthermore, by defining $Keyword \xleftarrow{Q} Author = S \cdot A^\circ$ its possible to identify on which areas a given author has her/his papers cited,

$$k Q a = \langle \exists p : a A p : k S p \rangle$$

In words: “There is a paper p , written by a , cited by some other paper in the area of k ”.

However, these relations only allow the specification of *reachability* properties in the current Alloy implementation, due to \cup -idempotency as mentioned previously. What if they were perceived from the *Mat* point-of-view? Then S becomes $K \times K \cdot C^\circ$, and thus:

$$k S p = \text{if } (k K p) \text{ then } \langle \sum q : p C q \wedge k K q : 1 \rangle \text{ else } 0$$

This means that S is able to quantify *how many* papers cite paper p in the same area k . In the new setting, Q becomes

$$\begin{aligned} k Q a &= \langle \sum p : a A p : k S p \rangle \\ &= \langle \sum p : a A p \wedge k K p : \langle \sum q : p C q \wedge k K q : 1 \rangle \rangle \\ &= \langle \sum p, q : a A p \wedge k K p \wedge p C q \wedge k K q : 1 \rangle \end{aligned}$$

which relates the author a to the *number* of citations she/he received in the field k .

Simply by perceiving this model through *Mat*, bibliometrics become instantly extracted from it! In possession of higher expressiveness, more sophisticated analyses can be performed via operators that one did not have access to in the relational model. For instance, one can reason about the overall contribution of an author in a particular area by defining, $\text{Keyword} \xleftarrow{Z} \text{Author} = \frac{Q}{S.T}$

$$k Z a = \frac{\langle \sum p : a A p : k S p \rangle}{\langle \sum q :: k S q \rangle}$$

which measures the percentile of author a in the field k .

We just had a sneak peek at the advantages of lifting a relational model to the quantitative matrix realm, in particular, by also considering matrices under \times , the Hadamard product.¹ Several other operators can potentially be included, leading to whole sets of unique properties that could not be expressed otherwise.

The main aim of this project is to hopefully improve the Alloy Analyzer so as to be capable of deriving more information from existing models – as illustrated with relations S and Q above –, while also supporting new kinds of operators, so that additional relations (now matrices) like Z can be seamlessly constructed on old and new models alike.

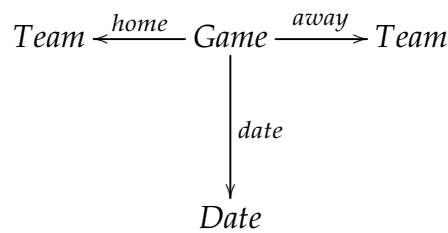
4.1.2 Football Championship

Consider the prospect of devising a game scheduler for a football championship. Clearly, a valid tournament schedule will have to meet the following requirements:

¹ This corresponds to relation intersection at relation level, as seen in Section 2.2.

- (a) The same team cannot play two games on the same day;
- (b) Every team has to play with every other, but not with itself;
- (c) For each home game there is another game away involving the same two teams.

This problem was originally introduced in *Program Design by Calculation* course and showcases the capabilities of Alloy, with which one is able to determine multiple game agendas satisfying all the conditions imposed. Appendix A.2.1 contains a proposed Alloy specification for this problem, based on the type diagram that follows:



Furthermore suppose that it is also required for the model to be able to track the number of wins and losses of each team. The initial problem could then be extended to consider a relation *History* which allows repeated arcs, like this dissertation aims to achieve in Alloy by the end of this project.

$$\{Win, Lose\} = Result \xleftarrow{History} Team$$

Figure 4.1 pictures an example of such a relation, where the number associated with each arc represents the number of occurrences in the relation, for instance, $(t_1, Lose)$ appears 6 times in the relation, meaning that t_1 , in total, lost 6 games during the tournament.

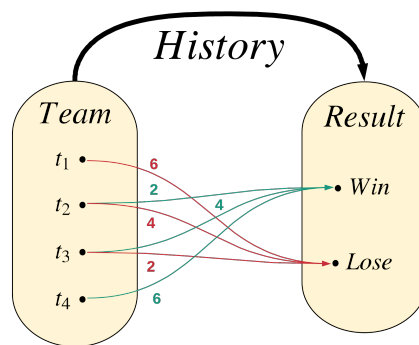


Figure 4.1: Example of the relation *History*.

Since, at this point, *History* cannot be precisely specified in Alloy, let us explore how this version of the problem can be modeled from an SMT point-of-view, in order to bring this work one step closer to its goal.

The specification will then be written in the SMT-LIB language, with respect to the theory of integers, using QF_UFLIA – *unquantified linear integer arithmetic with uninterpreted sort and function symbols* – logic, and solved with resort to CVC4. Starting with the original problem, the following integer constants shall be declared:

g_{had} – Number of games played on team h 's home, against team a , on day d .

Now, to accomplish the initial requirements, the necessary constraints are specified:

- Non-negativity constraints

$$g_{had} \geq 0$$

- No team can play two games on the same date

$$\forall x \in Team, d \in Date. \sum_{y \in Team} g_{xyd} + g_{yxd} \leq 1$$

- All teams play against each other but not against themselves

$$\forall x \in Team. \sum_{d \in Date} g_{xxd} = 0$$

$$\forall x, y \in Team. x \neq y \Rightarrow \sum_{d \in Date} g_{xyd} \geq 1$$

- For each home game there is another game away involving the same two teams

$$\forall x, y \in Team. \sum_{d \in Date} g_{xyd} = \sum_{d \in Date} g_{yxd}$$

Next, in order to deal with the championship results, the required constants and constraints over them will be specified:

w_i – Number of games won by team i .

l_i – Number of games that team i lost.

w_{ij} – Number of games that team i won against team j .

l_{ij} – Number of games that team i lost against team j .

- Non-negativity constraints

$$w_{ij} \geq 0 \wedge l_{ij} \geq 0$$

- The total number of games played by each team is equal to the sum of all their wins and losses

$$\forall i \in Team. w_i + l_i = \sum_{j \in Team} \sum_{d \in Date} g_{ijd} + g_{jid}$$

- The total number of games won (resp. lost) by each team is the same as the amount of games won (resp. lost) against every team

$$\forall i \in Team. w_i = \sum_{j \in Team} w_{ij} \wedge l_i = \sum_{j \in Team} l_{ij}$$

- The number of times that team i won against team j is the same as the number of times that team j lost against team i

$$\forall i, j \in Team. w_{ij} = l_{ji}$$

- Games between each team

$$\forall i, j \in Team. w_{ij} + l_{ij} = \sum_{d \in Date} g_{ijd} + g_{jid}$$

An excerpt of the full specification for $Team = [1..4]$ and $Date = [1..6]$ can be found in the Appendix [A.2.2](#). After filling the assertion stack with these declarations and constraints, by adding the command (`check-sat`), as explained in Section [3.1.1](#), to the stack, it requests CVC4 to solve the model with respect to the state of the assertion stack at this point, providing the SAT/UNSAT/UNKNOWN judgement afterwards. Given a SAT/UNKNOWN response, then the command (`get-model`) can be used to obtain the solution associated with the previous (`check-sat`) command, containing the values assigned to each function symbol within the specification. Then, if the SMT specification, together the commands to solve and extract the results, is stored in a `championship.smt2` file, CVC4 can be used to analyse the SMT problem as follows:

```
$ cvc4 championship.smt2
```

for which the following solution was obtained:

```

sat                ; result of check-sat
(model             ; get-model displays the solution
(define-fun g111 () Int 0)
(define-fun g112 () Int 0)
(define-fun g113 () Int 0)
(define-fun g114 () Int 0)
(define-fun g115 () Int 0)
(define-fun g116 () Int 0)
(define-fun g121 () Int 0)
(define-fun g122 () Int 0)
(define-fun g123 () Int 1)
(define-fun g124 () Int 0)
(define-fun g125 () Int 0)
(define-fun g126 () Int 0)
...
(define-fun l21 () Int 0)
(define-fun l23 () Int 2)
(define-fun l24 () Int 2)
(define-fun l31 () Int 0)
(define-fun l32 () Int 0)
(define-fun l34 () Int 2)
(define-fun l41 () Int 0)
(define-fun l42 () Int 0)
(define-fun l43 () Int 0)
)

```

characterized as follows:

- The constants g_{123} g_{136} g_{142} g_{215} g_{232} g_{246} g_{314} g_{321} g_{345} g_{411} g_{424} g_{433} were set at 1, while the remaining g_{had} had their value at 0
- $w_1=0$ $l_1=6$ $w_2=2$ $l_2=4$ $w_3=4$ $l_3=2$ $w_4=6$ $l_4=0$

corresponding to the calendar and win/loss history pictured in Table 4.1.

Dates	home	away
Date 1	Team 3 Team 4	Team 2 Team 1
Date 2	Team 1 Team 2	Team 4 Team 3
Date 3	Team 2 Team 4	Team 1 Team 3
Date 4	Team 3 Team 4	Team 1 Team 2
Date 5	Team 1 Team 3	Team 2 Team 4
Date 6	Team 1 Team 2	Team 3 Team 4

(a) Championship calendar.

Participants	Wins	Losses
Team 1	0	6
Team 2	2	4
Team 3	4	2
Team 4	6	0

(b) Performance of each team in this tournament.

Table 4.1: Example of a football championship obtained using CVC4.

As illustrated, it is possible to obtain one championship schedule satisfying all the prerequisites desired, alongside a possible outcome for that tournament. In particular, the assignments highlighted in the Table 4.1b can be used to build precisely the relation *History* previously exemplified in Figure 4.1. Furthermore, this model is capable of finding *all* the possible valid game calendars as well as *all* the potential results for the configuration considered (four teams participating and six possible dates) and can be easily modified to accommodate to other *Team* and *Date* values.

4.1.3 Sprinkler

Entry [Bayesian Network – Wikipedia](#) gives as example of *Bayesian network* a probabilistic model of a sprinkler wetting grass, whose activation also depends on it raining or not, pictured in Figure 4.2. A *Bayesian network* corresponds to a *directed acyclic graph* with edges defining conditional dependencies and nodes representing unique random variables.

By interpreting each conditional probabilistic table associated with each node as a left-stochastic matrix one can reason algebraically using the concepts described before in Section 2.2.

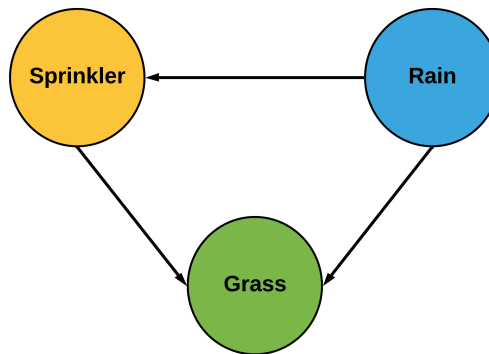


Figure 4.2: Bayesian network for the Rain, sprinkler and grass problem.

$$R \xleftarrow{rain} 1 = \begin{bmatrix} 0.80 \\ 0.20 \end{bmatrix} \quad S \xleftarrow{sprinkler} R = \begin{bmatrix} 0.60 & 0.99 \\ 0.40 & 0.01 \end{bmatrix}$$

$$G \xleftarrow{grass} S \times R = \begin{bmatrix} 1.00 & 0.20 & 0.10 & 0.01 \\ 0 & 0.80 & 0.90 & 0.99 \end{bmatrix}$$

For instance, the probability of the grass being wet, $P(g = 1)$, can be determined through:

$$1 \xleftarrow{wet} G \xleftarrow{grass} S \times R \xleftarrow{sprinkler \vee id} R \xleftarrow{rain} 1$$

$P(g=1)$

where $1 \xleftarrow{wet} G = [0 \ 1]$.

After solving the vector-matrix multiplications, it can be observed that there is a 44.838% chance of the grass being wet for this network configuration.

$$[0 \ 1] \cdot \begin{bmatrix} 1.00 & 0.20 & 0.10 & 0.01 \\ 0 & 0.80 & 0.90 & 0.99 \end{bmatrix} \cdot \begin{bmatrix} 0.6 & 0 \\ 0 & 0.99 \\ 0.4 & 0 \\ 0 & 0.01 \end{bmatrix} \cdot \begin{bmatrix} 0.80 \\ 0.20 \end{bmatrix} = [0.44838]$$

Could PRISM be used to model this scenario and verify probabilistic properties over the network?

From all the different model types provided, *Discrete-Time Markov Chain* seems adequate to specify the *Bayesian network* in question. Moreover, every node corresponds to a *module*, each with a *variable* that identifies its state, having one of three possible values. For example, for *rain* the variable *r* is considered, declared as follows:

```

module rain
  r : [0..2] init 2;
  ...

```

such that, each value is characterized as:

- 2 represents the initial state;
- 0 indicates that it is not *raining* (*off* and *dry* for *sprinkler* and *grass*, respectively);
- 1 means that it is currently *raining* (respectively *on* and *wet* for the *sprinkler* and *grass* modules);
- 0 or 1 are both terminal states.

Every stochastic matrix is then encoded with resort to *commands*, one for each column of the matrix, e.g. “If the sprinkler is on and it is not raining, then the grass is wet with 90% probability or dry with 10% chance” is represented by $[] \text{ s}=1 \ \& \ \text{r}=0 \ \& \ \text{g}=2 \ \rightarrow \ 0.9 : (\text{g}' = 1) + 0.1 : (\text{g}' = 0)$.

In the end, *rain* is specified by the following process:

```

module rain
  r : [0..2] init 2;

  [ ] r=2 -> 0.8 : (r' = 0) + 0.2 : (r' = 1);
  [ ] r=0 | r=1 -> (r'=r); //0 and 1 are terminal states
endmodule

```

The full PRISM specification of this network can be seen in the Appendix [A.3.1](#).

At this point, one may specify PCTL properties about the model, such as:²

- “What is the probability of the grass being wet?”

$P=? [F \text{ g}=1]$

- “What are the odds of the sprinkler being on, knowing that it is not raining?”

$P=? [F \text{ s}=1 \ \& \ \text{r}=0] / P=? [F \text{ r}=0]$

- “Probability of rain, assuming the grass is wet.”

$P=? [F \text{ r}=1 \ \& \ \text{g}=1] / P=? [F \text{ g}=1]$

Property: $P=? [F g=1]$
Defined constants: <none>
Method: Verification
Result (probability): 0.44838 (value in the initial state)

Figure 4.3: Calculating the probability of the grass being wet using PRISM.

Figure 4.3 shows the result obtained for the first property, which specifies $P(g = 1)$ and, as expected, it coincides with the same value obtained previously.

In the end, a *Bayes network* was successfully modeled by taking advantage of PRISM. Continuing with the study of the capabilities of this tool, consider the following situation: what if the *sprinkler's* conditional probabilistic table was unknown? That is,

$$S \xleftarrow{\text{sprinkler}} R = \begin{bmatrix} 1 - s10 & s01 \\ s10 & 1 - s01 \end{bmatrix}$$

with s_{yx} representing the probability of the sprinkler being on/off ($s = y$) given that it is or it is not currently raining ($r = x$).³

It would be interesting to be able to answer questions such as “Is there a *sprinkler* for which the probability of the grass being wet is higher than 90%?”, i.e., verify if exists a *sprinkler* causing $P(g = 1) > 0.9$. Therefore, one should start by adapting the previous PRISM specification to the current setup:

1. New probabilistic constants declaration;

```
const double s10;
const double s01;
```

2. *sprinkler* module readjustment – writing the probabilities with respect to the constants.

```
[ ] r=0 & s=2 -> s10 : (s' = 1) + (1-s10) : (s' = 0);
[ ] r=1 & s=2 -> (1-s01) : (s' = 1) + s01 : (s' = 0);
```

Since the model now contains unknown constants, the user needs to assign a value for each of them before PRISM is able to check the validity of the property, as shown previously in Figure 4.3. In particular, by setting $s10 = 0.4$ and $s01 = 0.99$ the current model is reduced to the earlier version and thus, the

² Conditional probability definition considered: $P(A|B) = \frac{P(A \cap B)}{P(B)}$.

³ Note that $s00 = 1 - s10$ and $s11 = 1 - s01$ since the columns of a left-stochastic matrix must add up to 1.

same result is obtained for that property. However, by taking advantage of *experiments*, the tool is able to verify multiple instances of the same model according to a range of values set by the user, in the same manner presented in the Figure 4.4, which checks the property for all the combinations of possible values of s_{10} and s_{01} , each starting at 0 and going up to 1 with step 0.01.

Name	Type	Single Va...	Range:		
			Start	End	Step
s10	double	<input type="radio"/>	0	1	0.01
s01	double	<input type="radio"/>	0	1	0.01

Figure 4.4: Constants range definition.

Then, $P(g = 1) > 0.9$ is specified in PCTL as $P > 0.9 [F g=1]$ and an experiment is executed with the constants varying according to that range of values, as illustrated in Figure 4.5.

Property	Defined Constants	Progress
$P > 0.9 [F g=1]$	$s_{10}=0.0:0.01:1.0, s_{01}=0.0:0.01:1.0$	10201/10201 (100%)
	Status	Method
	Done	Verification

Figure 4.5: Verifying $P(g = 1) > 0.9$ with *experiments*.

Table 4.2 presents a few results of the verification instances of this experiment.

s10	s01	Result
0.98	0.98	<i>false</i>
0.98	0.99	<i>false</i>
0.98	1.0	<i>false</i>
0.99	0.0	<i>true</i>
0.99	0.01	<i>true</i>
0.99	0.02	<i>true</i>

Table 4.2: Extract of the experiment results.

As can be observed, $S \xleftarrow{\text{sprinkler}_F} R = \begin{bmatrix} 0.02 & 1 \\ 0.98 & 0 \end{bmatrix}$ represents a *sprinkler* for which the probability of the grass being wet is not higher than 90%, while $S \xleftarrow{\text{sprinkler}_T} R = \begin{bmatrix} 0.01 & 0.02 \\ 0.99 & 0.98 \end{bmatrix}$ showcases a *sprinkler* where that is the case.

Alternatively, parametric model checking can also be used to determine such *sprinklers*, for example, if `sprinkler.prism` and `sprinkler.props` are the files containing the PRISM model and the property $P > 0.9 [F g=1]$, respectively, then parametric model checking over them can be performed with 0.01 parameter precision by executing:

```
$ prism sprinkler.prism sprinkler.props
  -param s10=0:1,s01=0:1 -paramprecision 0.01
```

which arrived at the results presented in Table 4.3, from which one is able to extract *sprinklers* for which $P(g = 1) > 0.9$ by the various combinations arising from the interval where the result is *true*. In particular the previous *sprinkler_F* and *sprinkler_T* considered from the experiment arrive at the same result using parametric model checking. To be noted that the solutions obtained are with respect to a certain precision, in this case 0.01, meaning that some of them could not be entirely accurate, and naturally, by executing parametric model checking given a further precise value, the more exact the results obtained.

s10	s01	Result
[0.5, 0.96875]	[0.0, 0.5]	<i>false</i>
[0.984375, 1.0]	[0.0, 0.15625]	<i>true</i>
[0.96875, 0.984375]	[0.1875, 0.3125]	<i>false</i>
[0.96875, 0.984375]	[0.34375, 0.4375]	<i>false</i>
[0.0, 0.5]	[0.0, 1.0]	<i>false</i>
[0.5, 1.0]	[0.5, 1.0]	<i>false</i>

Table 4.3: Parametric model checking results.

In conclusion, PRISM proved itself as powerful probabilistic model checker that is seamlessly compatible with the Alloy extension theoretical framework, and whose stochastic and nondeterministic capabilities seem adequate to the Alloy characteristics and needs, becoming a promising candidate for the probabilistic component of this dissertation.

4.2 PROBLEM

In order to implement quantitative reasoning in the Alloy framework, one must establish connections between the first and the necessary tools, capable of handling this type of constraints, as well as coordinating their interactions accordingly.

Initially, it is necessary to extend the underlying Boolean matrix representation to support both $Mat_{\mathbb{N}_0}$ and LS matrices, which implies implementing these concepts in Kodkod. Therefore, one will explore how to represent a *Kodkod problem* containing:

- Numeric constraints, their matrix representation and further feed the resulting logical formula to an SMT Solver. Moreover, an adequate theory needs to be determined and passed to the solver alongside the specification at hand in order to make the solving feasible.
- Probabilistic properties, which requires encoding the corresponding matrices into suitable PRISM modules and then take advantage of the probabilistic model checking features provided by PRISM to

determine valid distributions, check the satisfiability of properties in a qualitative manner, calculate unknown probabilities, and so on, depending on the class of assertions being reasoned about.

After successfully being able to solve such kind of Kodkod problems, by consequence, Alloy will be very close to supporting quantitative specifications. All there is left to do is adapt the syntax and semantics of the Alloy language affected by the shift to the quantitative realm, potentially extending the language to further include operators that could not be included before due to the lack of expressiveness, like multiplication and division between numeric values over both \mathbb{N}_0 and $[0, 1]$ domains. Furthermore, the Alloy Analyzer needs to be modified to produce adequate Kodkod problems from quantitative Alloy specifications, as well as being able to interpret the new type of resulting Kodkod instances. Finally, the features provided by the analyzer, such as instance visualization and interactive evaluation, have to be adjusted to such models.

4.3 PROPOSED APPROACH - SOLUTION

Figure 4.6 presents the current version of Alloy's underlying course of action when finding valid instances for a given specification or check for counterexamples to some property.

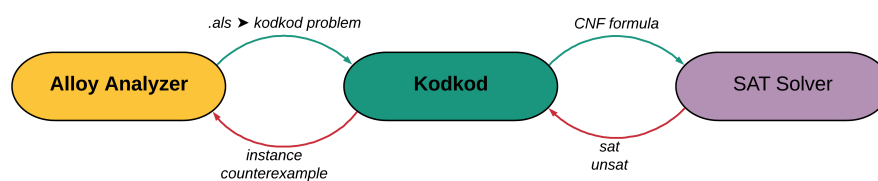


Figure 4.6: Current Alloy Architecture.

4.3.1 System Architecture

To accomplish the quantitative extension to the Alloy framework as described previously, the implementation of the workflow illustrated in Figure 4.7 is expected. In case the increased expressiveness is not required or desired for a specific specification, the conventional SAT solvers will still be used, according to the existing implementation. When interested in performing implicit quantitative analysis of existing Alloy models or model checking Alloy specifications containing explicit numeric quantitative constraints, SMT solvers will function instead. Otherwise, when before a probabilistic setting, the framework will take advantage of PRISM's capabilities.

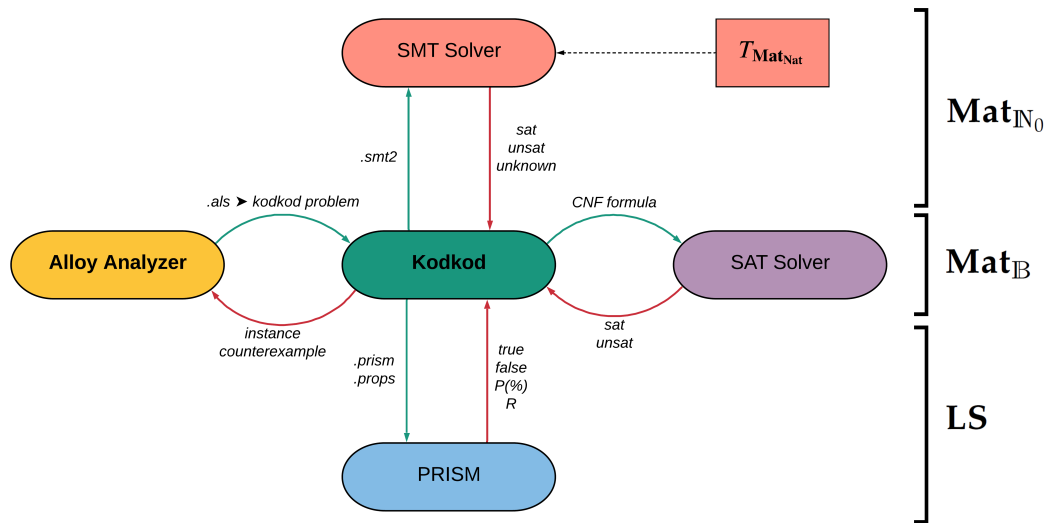


Figure 4.7: Proposed Quantitative Alloy Architecture.

4.4 SUMMARY

This chapter presented various case studies which benefit from quantitative analysis, each with distinct characteristics, from explicitly taking advantage of the increased expressiveness provided in the theoretical base as was the case of the football championship case study, implicitly extract quantitative data from just changing category like in the bibliographic system model, to the application of probabilistic model checking techniques when encoding and reasoning about left-stochastic matrices in the *bayesian network* example. Furthermore, the main tools which will take part in building the Alloy extension had their capabilities evaluated by applying them to these case studies. At last, the workflow designed for such quantitative extension was illustrated and characterized.

QUANTITATIVE KODKOD

Having laid the theoretical groundwork necessary to accomplish the quantitative extension of Alloy, this chapter starts by exploring the challenges and implementation alternatives that arise when trying to accommodate Alloy and, by consequence, Kodkod to the quantitative semantics of typed linear algebra.

The next step is to design the structures that better suit all the implementation decisions made, the requirements to properly handle both quantitative scenarios ($Mat_{\mathbb{N}_0}$ and LS), as well as take into account the existing structures, used to execute qualitative analysis.

Afterwards, alongside the quantitative extension of Kodkod's abstract syntax, the management routine of the structures established to handle quantitative Kodkod problems described by first-order relational logic is detailed, defining a correspondence between the two.

5.1 ATOMS AND RELATIONS

As stated previously, an Alloy model is represented through *atoms* and *relations*. Furthermore, its atoms arise from the *signatures* specified (together with the scope considered for a given evaluation), i.e., they are the elements within the *sets* – unary relations – present in the model, while relations with arity greater or equal to 2 are declared through *fields* inside these *signatures*, over the model's sets. At **Kodkod** level they are indistinguishable, being represented by the same kind of structure, a Boolean matrix. In particular, *signatures* correspond to vectors while *fields* are defined by matrices with more than one dimension.

Now, when extending Boolean matrices into the quantitative context, by allowing natural numbers or probabilities as cell kinds, every tuple in a relation gets a meaningful *weight* associated to it, beyond the two possible values in the Boolean setting – 1 when the tuple is present in the relation, and 0 otherwise. Therefore, one has to make an explicit distinction between the previously highlighted concepts, due to the impact of increased expressive power, for instance, when declaring a relation $R : A \rightarrow B$ using Alloy syntax as illustrated in Figure 5.1.

```
sig B{}
sig A{ R : set B }
```

Figure 5.1: Declaring $R : A \rightarrow B$ in Alloy.

This model has two signatures A and B , and R as a field of A . Simply put, in the corresponding Kodkod problem, R is then specified through the constraint:¹

$$R \subseteq A \times B$$

Since the vectors representing A and B are now numeric matrices, their atoms will *also* have an arbitrary weight associated. If $A = \{(A_0, 3), (A_1, 4)\}$ and $B = \{(B_0, 2)\}$, then $A \times B = \{(A_0 \rightarrow B_0, 6), (A_1 \rightarrow B_0, 8)\}$ and thus, $(A_0 \rightarrow B_0, 3)$ is a valid tuple of R but $(A_0 \rightarrow B_0, 7)$ is not.

Having *signatures* represented by numeric matrices adds an extra layer of complexity that is undesirable, as the acceptable weight for each tuple of a given relation will always have an upper bound with respect to the weight of each atom of the relation type sets – varying from instance to instance within a given scope – as well as depend on the model’s constraints, making the model harder to handle and analyse. Moreover, allowing each tuple to have their weight unbounded by default, while still respecting the relation type, seems a far more desirable and expected approach.

Further to this, having atoms with an arbitrary weight associated by itself violates its definition, in particular, they can no longer be considered as *indivisible* and thus, abstracting weights from atoms is deemed necessary.

Taking all the points above into account, in the proposed implementation, *signatures* and their atoms will **still** be represented through Boolean matrices, that is, no Kodkod unary relation will be able to have tuples with weight outside of the values $\{0, 1\}$.

It should be noted that, in case the user needs to explicitly describe an unary relation whose tuples may have non-Boolean valued weights associated, it is still possible to do so, for example, $S : A$ can be specified as $S : () \rightarrow A$, corresponding to the Alloy code specified in Figure 5.2.

```
sig A{}
one sig Unit{ S : set A }
...
let S' = Unit.S | ...
```

Figure 5.2: Specification of the unary set $S : A$ in a quantitative setting.

Therefore, in case $S = \{(() \rightarrow A_0, 3), ({} \rightarrow A_1, 1), ({} \rightarrow A_2, 10)\}$, through composition the desired set can be accessed as $S' = S \cdot () = \{(A_0, 3), (A_1, 1), (A_2, 10)\}$, like Figure 5.3 shows.

¹ The proper Kodkod constraint generated in a quantitative context will be seen in greater detail in Section 6.2.

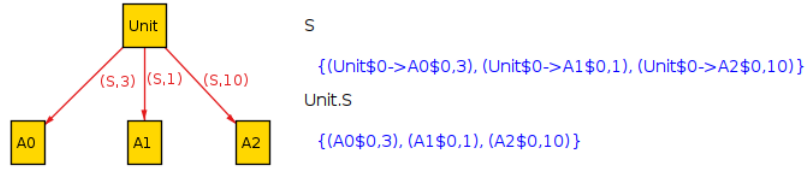


Figure 5.3: An instance from the model displayed in Figure 5.2.

5.2 BOOLEANS GO QUANTITATIVE

Given that the elements of a Boolean matrix are of type \mathbb{B} , one can consider \mathbb{F} as an extension to the quantitative realm, where $\mathbb{F} = \mathbb{N}_0 + [0, 1] \subseteq \mathbb{R}$, i.e., the type of the cells of a numeric matrix depend on the kind of setting at hand, representing either a matrix of $Mat_{\mathbb{N}_0}$ or LS .

Moreover, a correspondence between the two must be established. If b_i is a Boolean value and x_i is its numeric counterpart (or vice versa), then their relation is precisely specified as:

$$b_i = x_i > 0$$

Considering that in this context, x will either be a natural number or a probability, then

$$b_i = false \equiv x_i = 0.$$

In order to handle both representations, the operators *lift* and *drop* can be defined, analogously to those studied in the Section 3.2.1, as follows:

$$\begin{array}{ccc}
 & \xrightarrow{\text{lift}} & \\
 \mathbb{B} & & \mathbb{F} \\
 & \xleftarrow{\text{drop}} &
 \end{array} \tag{5.1}$$

$$\text{lift}(b_i) = \begin{cases} \text{trueVariable}(i) & \text{if } b_i \\ 0 & \text{otherwise} \end{cases} \qquad \text{drop}(x_i) = \begin{cases} \text{true} & \text{if } x_i > 0 \\ \text{false} & \text{if } x_i = 0 \end{cases}$$

with $\text{trueVariable}(i) = x_i$ constrained by $x_i > 0$. As can be seen, these operations are not isomorphic, in particular, *drop* results in information loss. This is expected, since *lift* increases expressiveness and vice versa.

NUMERIC NEGATION Now, one can define the negation of a numeric value by taking advantage of these operators.

$$\begin{aligned}
 \neg_{\mathbb{F}} &: \mathbb{F} \rightarrow \mathbb{B} \\
 \neg_{\mathbb{F}} &= \neg \cdot \text{drop} \\
 \neg_{\mathbb{F}} x &= x \leq 0
 \end{aligned}
 \tag{5.2}$$

QUANTIFIED EXPRESSION A *quantified expression* $Q \ e$, that is, a constraint over the expression e with respect to the *quantifier* or *multiplicity keyword* Q , highlighted previously in the Tables 2.5 and 2.6 respectively, must be properly re-defined with respect to the new lifted values.

At Kodkod level, the expression, which will be represented by a matrix M , is submitted to an adequate formula associated with the keyword considered from $\{\text{some}, \text{one}, \text{none}, \text{no}\}$ (keywords `set` and `all` do not need to be explicitly handled at this stage, since `set` does not constraint e in any meaningful way and `all e` has no meaning).

In regards to the `some` and `no`, their constraints are pretty straightforward, being similar to their qualitative counterpart. Analogously to the numeric negation defined previously, quantitative `some` can be obtained by transforming M into a Boolean matrix using *drop* and applying the qualitative `some`. Quantitative `no` could simply be defined as the negation of `some` or by checking if it corresponds to the null matrix (`none`), however, by taking advantage of the newly acquired expressiveness, in this implementation it will be specified in a more succinct way: the sum of every element of M must be equal to 0.

$$\text{some } M \equiv \bigvee_i M[i] > 0 \qquad \text{no } M \equiv \sum_i M[i] = 0$$

When defining `none` and `one` in a quantitative context, there is more than a sole possible interpretation due to the increase in expressiveness. For instance, by following the same line of thought as the previous definitions of $\neg_{\mathbb{F}}$ and `some`, simply taking M as its Boolean matrix correspondence using *drop*, and using the current definition of `one`, then `one e` will hold if and only if there is exactly one tuple present in e with *positive* weight, that is, a single element of M must have a positive value. For example, `one{(t,5)}` would evaluate to *true*. Thus, since `none = one || none`, `none e` would now mean that e would have no tuples, or a single tuple with a positive value associated. However, one can also argue that `one e` should only be satisfied if and only if e contains one tuple precisely with weight 1. Therefore, `none e` would be *true* when e possesses no tuples or one tuple with value 1 associated, i.e., the sum of all cells of M would be less or equal to 1.

As the latter interpretation is the expected one at first glance, besides being more precise, it is also easier to handle by users already experienced with Alloy, as this definition behaves fundamentally the same as the qualitative version, e.g., a function $f : A \rightarrow B$ would be declared as `sig A{ f : one B }` in both the original Alloy and this quantitative extension, therefore it will be the one put into effect during development.

$$\text{one } M \equiv \sum_i M[i] = 1$$

$$\text{!one } M \equiv \sum_i M[i] \leq 1$$

To be also noted that the definition initially proposed can still be achieved in practice by taking advantage of `drop2`, for instance, through encodings like `!one (drop e)`.

LOCAL VARIABLES Specifying quantification formulas or building relations by comprehension involves handling, respectively, atoms or tuples locally as variables. Shifting into a quantitative setting associates a numeric value to those variables. Since quantification constraints handle atoms, these will always have weight 1, as imposed by their definition, i.e., for every *signature* A , all $a : A \mid \text{one } a$ *always* holds. Now, when specifying the tuple's form by comprehension, there might be multiple possible weights associated that hold for the formula considered, even with respect to a particular instance from an already solved model. This extension will then convention that every tuple built by comprehension will be constrained to have weight 1, meaning that $\{D \mid F(D)\}$ will result in an empty set if there is no tuple of the form D with the value 1 associated that satisfies F , even if the same tuple with a weight strictly different than 1 would satisfy the formula considered.

5.3 NUMERIC STRUCTURES

Taking a more in-depth look at the Boolean structures present in the underlying Kodkod implementation, it can be noted that each Boolean matrix is represented through a **sparse sequence**, that is, a sequence that may or may not have contiguous *flat indices*, storing only non-zero (false) values. In particular, a N -ary relation R with scope $[L, U]$, delimited by the lower bound L and the upper bound U , over the universe $\mathcal{A} = \{a_0, a_1, \dots, a_k\}$ is described by a Boolean sparse-matrix M specified as follows (Torlak and Jackson, 2007):

$$M[i_1, \dots, i_N] = \begin{cases} \text{true} & \text{if } (a_{i_1}, \dots, a_{i_N}) \in L \\ \text{freshVar}() & \text{if } (a_{i_1}, \dots, a_{i_N}) \in U - L \\ \text{false} & \text{otherwise} \end{cases}$$

where $i_1, \dots, i_N \in [0, k]$ and `freshVar` declares a fresh Boolean variable. Furthermore, all relations are encoded as matrices where every dimension has size $|\mathcal{A}|$ – binary relations are represented by square matrices, and so on – and therefore, an element of M in the position $[i_1, \dots, i_N]$ is mapped into the *flat index* $\sum_{j=1}^N (i_j \times |\mathcal{A}|^{N-j})$ of its respective sparse sequence (Torlak and Jackson, 2006).

$$M : \underbrace{|\mathcal{A}| \times |\mathcal{A}| \times \dots \times |\mathcal{A}|}_N$$

² The addition of `drop` to the Alloy language is described in Section 6.2.

This way, any tuple over \mathcal{A} is precisely uniquely identified by its index.

In the process of transforming a Kodkod problem into CNF formulæ to be solved by a SAT Solver, the original specification is described by Boolean structures such as the previous Boolean matrices, which are handled by assembling **Compact Boolean Circuits** (CBCs), ideal to represent an equivalent Boolean encoding, as they are able to detect common equivalent sub-formulas with respect to a certain degree, reducing the size of the final problem and thus, decreasing solving times.

Such kind of circuit is characterized by a partially canonical, directed, acyclic graph (V, E, d) whose vertices V are either logical gates $V_{op} = V_{\wedge} \cup V_{\vee} \cup V_{\neg}$ or leaves (e.g. elements of Boolean matrices) $V_{leaf} = V_b \cup \mathbb{B}$, where V_b contains a Boolean variable. d corresponds to the circuit *compaction depth* and together with an equivalence relation over V they describe the circuit's degree of canonicity. Figure 5.4 showcases an example of a non-compact Boolean circuit and its CBC equivalent for $d = 2$.

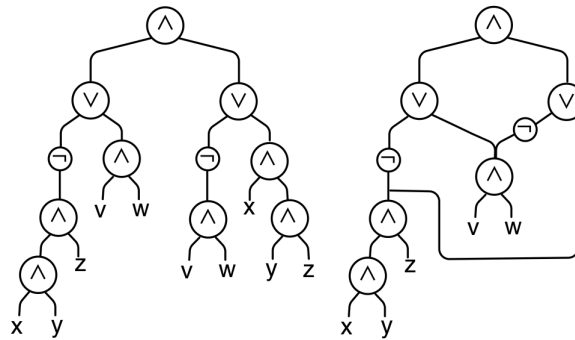


Figure 5.4: A non-compact Boolean circuit on the left and the corresponding CBC ($d = 2$) in the right (Torlak and Jackson, 2007).

In order to achieve quantitative analysis, first these structures must be extended to be able support problem encoding of increased expressiveness. In particular, Boolean matrices will no longer be used, but **numeric matrices** instead, whose elements belong to \mathbb{F} and are related to Boolean values exactly as described previously in Section 5.2, by also taking advantage of sparse sequences. Hence, a relation R in the same conditions imposed previously when characterizing Boolean matrices, in a quantitative setting will be represented by a numeric sparse-matrix M whose definition will depend on its arity, as discussed formerly in Section 5.1, whether R represents an Alloy *signature* or *field*.

- $N = 1$ (*Signature*)

$$M[i] = \begin{cases} 1 & \text{if } a_i \in L \\ \text{freshVar}(\{0, 1\}) & \text{if } a_i \in U - L \\ 0 & \text{otherwise} \end{cases}$$

where $i \in [0, k]$ and $\text{freshVar}(\{v_1, v_2, \dots, v_n\})$ declares a fresh numeric variable x such that $x \in \{v_1, v_2, \dots, v_n\} \subseteq \mathbb{F}$.

- $N > 1$ (*Field*)

$$M[i_1, \dots, i_N] = \begin{cases} trueVariable() & \text{if } (a_{i_1}, \dots, a_{i_N}) \in L \\ freshVar() & \text{if } (a_{i_1}, \dots, a_{i_N}) \in U - L \\ 0 & \text{otherwise} \end{cases}$$

The present definition is obtained by lifting the former Boolean matrix specification and consequently $i_1, \dots, i_N \in [0, k]$, *trueVariable* and *freshVar* declare a new variable $x \in \mathbb{F}$ with *trueVariable* also imposing that $x > 0$.

Likewise, $|\mathcal{A}|$ is the size of all N dimensions of M and so, the same flat indexing mechanism is taken into account when addressing R 's tuples: if the position of the sparse-matrix identified by the flat index of $[i_1, \dots, i_N]$ contains the numeric value w then $M[i_1, \dots, i_N] = w$ meaning that the tuple $(a_{i_1} \rightarrow \dots \rightarrow a_{i_N}, w)$ is present in R as long as $w \neq 0$.

To manage this new kind of matrices, CBCs are no longer adequate, meaning that a new type of structure must be established. Nonetheless, it is important to note that Boolean reasoning cannot be completely discarded either, as a Kodkod problem will still be described by a first-order logic formula and thus, the numeric structures considered must be flexible enough to naturally handle the numeric component while also being able to deal with Boolean constraints when convenient. Therefore, in this implementation, a similar approach to Boolean circuits will be taken, making sure that the already existing Boolean structures can be taken advantage of, even in a quantitative setting, as in, if given a CBC as input, the extension should be able to process it so that analysis can be performed from a quantitative point-of-view using tools like SMT Solvers and so on. Then, the numeric structures that arise from the given Kodkod problem shall take the form a circuit. For that, the current CBC definition must be further enriched to include numeric values and expressive enough operators to support numeric matrices operations. Taking all the requirements mentioned previously into account, to reason over numeric structures a **Numeric Circuit** (NC) will be used, defined by a directed, acyclic graph (V, E) , with vertices V corresponding either:

LEAVES $V_{leaf} = V_b \cup \mathbb{B} \cup V_x \cup \mathbb{F}$ a Boolean or numeric constant, a Boolean variable (V_b) or a numeric variable (V_x).

GATES $V_{op} = V_{\mathbb{B}} \cup V_{\mathbb{F}} \cup V_{\square}$

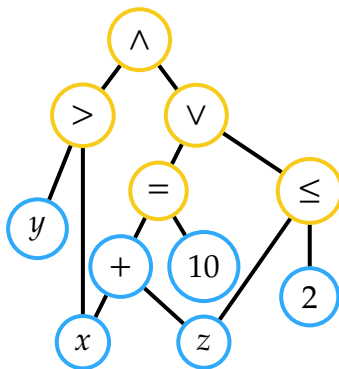
- Logical $V_{\mathbb{B}} = V_{\wedge} \cup V_{\vee} \cup V_{\neg}$
- Arithmetic³

$$V_{\mathbb{F}} = \begin{cases} V_{+} \cup V_{-} \cup V_{\times} \cup V_{/} \cup V_{mod} & \text{if } \mathbb{F} = \mathbb{N}_0 \\ V_{+} \cup V_{-} \cup V_{\times} \cup V_{/} & \text{if } \mathbb{F} = [0, 1] \end{cases}$$

³ If \mathbb{F} represents the natural numbers, then $/$ corresponds to the Euclidean division.

- Inequality $V_{\sqsubseteq} = V_{=} \cup V_{<} \cup V_{>} \cup V_{\leq} \cup V_{\geq}$

Numeric circuits encompass a mix of Boolean and arithmetic circuits/expression DAGs, augmented with inequality operators. Through NCs, one is now able to encode formulas like $y > x \wedge (x + z = 10 \vee z \leq 2)$ for $\mathbb{F} = \mathbb{N}_0$ as follows:



While this type of circuit is based on the concept of CBC – every CBC is a NC –, it should be highlighted that NC is *not* a proper extension of CBC, as it does not guarantee partial canonicity, that is, while they are structured in the same fashion, their assembly process will be different; a compaction depth is not part of a NC specification, meaning that shared sub-components of a circuit will not necessarily be detected.

5.4 SCOPE

The way of delimiting the extent of instances being analysed for a given Alloy command will remain unchanged in a quantitative setting, as well as the respective representation of the fixed scope in the derived Kodkod problem. After specifying the scope in the Alloy model, the number of atoms contained in each top-level signature will start at a given point (depending on the precision considered) and reach up to the ceiling imposed, which in turn shapes the boundaries of each Kodkod relation describing the model's signatures and fields.

Every relation $R : T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_N$ has **bounds** $[L, U]$ associated, where the lower bound L contains the tuples which *must* occur in R and the upper bound U consists of the tuples which *may or may not* appear in R for the instance to be a valid Kodkod model, and naturally, $L \cap U = L$.

Whether a tuple t of the form (t_1, t_2, \dots, t_N) with $t_1 \in T_1, t_2 \in T_2, \dots, t_N \in T_N$ is present in R , depends on the numeric sparse-matrix definition specified previously.

When $N = 1$, t is an atom, and therefore:

- if $t \in L$ then $t = 1$
- if $t \notin L \wedge t \in U$ then $t = 0 \vee t = 1$

- if $t \notin L \wedge t \notin U$ then $t = 0$, and thus, t can never be present in R of a Kodkod instance for the problem at hand with respect to this bounds

For $N > 1$, the constraints considered for $N = 1$ are relaxed:

- if $t \in L$ then $t > 0$ and thus, t must occur with a strictly positive value associated within \mathbb{F}
- if $t \notin L \wedge t \in U$ then $t \geq 0$, meaning that this tuple's weight depends only on the problem domain \mathbb{F} and the constraints specified in the original Alloy model
- if $t \notin L \wedge t \notin U$ then $t = 0$, like before R will not be allowed contain t

INTEGER SCOPE Due to the nature of the representation of integers originally implemented, in a qualitative setting, these are also bounded. Integers are contained in the *signature* `Int`, having each integer represented explicitly as an atom. Alloy's integer arithmetic works in the same fashion as *modular arithmetic*, and therefore the scope populates `Int` to achieve it.

Furthermore, the Kodkod engine handles integers and operations between them bitwise, representing each integer in *two's complement*, suitable to be processed by SAT solvers.

In the command `run{ ... }` for n `Int`, n expresses exactly the integer bitwidth to be considered in the two's complement representation, hence $\text{Int} = \{-2^{n-1}, \dots, 2^{n-1} - 1\}$ during the analysis of this command.

When shifting into the quantitative realm, this concept of integer scope is no longer useful, as the tools that will be used during solving possesses the capabilities to handle effectively "unbounded" integers that the SAT solvers lack.

Instead of discarding integer bounds altogether when performing quantitative analysis, they will be re-defined to provide the modeler another tool, acquiring finer control over the instances being studied.

When model checking a command like the previous in a quantitative setting over \mathbb{N}_0 , now n will dictate the **maximum** integer value that each tuple of every *field* in the model can assume, causing the previous scope definition to be extended, for $N > 1$:

- if $t \notin L \wedge t \in U$ then $0 \leq t \leq n$

For example, $R = \{(t, 10), \dots\}$ will not be a valid interpretation when solving `run{}` for 5 `Int`, but it may be for the command `run{}` for 20 `Int`.

If n `Int` is not explicitly stated in a command, then the tuple's weight will not be upper bounded by the scope, meaning that the former definition will be used.

Finally, if $F = [0, 1]$ then the integer scope will not have any interpretation in this context, being ignored during analysis, as the weight of each tuple now describes a probability.

5.5 NUMERIC CIRCUIT ASSEMBLY

In order to perform quantitative analysis using Kodkod, the latter has to be enhanced with the necessary notions discussed throughout this and former chapters, namely the *numeric structures* that support such kind of Kodkod problems and eventual new syntax to allow the modeling of quantitative concepts.

Thus, this section will go over the representation of those structures, their properties, and how they can be used in practice. Furthermore, a quantitative Kodkod problem will be characterized under a quantitative domain \mathbb{F} , including the extension of the Kodkod abstract syntax to allow for further kinds of quantitative constraints, and how the numeric structures are taken advantage of to support these new kinds of Kodkod problems.

NUMERIC STRUCTURES Each vertex of a Compact Boolean Circuit is represented through a `BooleanValue`, which is the elementary data type that composes the Boolean structures. A `BooleanValue` is either a `BooleanConstant` representing one of *true* and *false*, or a `BooleanFormula` to encode non-constant Boolean values such as `BooleanVariables` and the logical gates, showcased in Table 5.1.

The elements of a sparse-sequence associated with a Boolean matrix also correspond to `BooleanValues`. Both `BooleanValue` and `BooleanMatrix` are produced and managed by a `BooleanFactory` which, together with `CBCFactory` – a factory responsible for the assembly of the variable components of a CBC –, is used to build an adequate CBC associated with the Kodkod problem at hand accordingly.

Furthermore, the factory responsible for a particular solving instance attributes an integer *label* to every `BooleanValue` present, so that each of them is uniquely identified under such factory. Only the `TRUE` and `FALSE` constants have the same label, since they are shared with every factory. These labels possess meaningful characteristics and, alongside the labelling method adopted by a `BooleanFactory`, are associated to the `BooleanValues` so that further properties arise, meeting the conditions, together with caching mechanisms, necessary to effectively produce proper CBCs.

Numeric structures are built upon these concepts and thus, `BooleanValue` \leftarrow `NumericValue`⁴ appears as the main kind of data being handled in a quantitative setting, relating to \mathbb{F} the same way as `BooleanValue` is related to \mathbb{B} . As expected, a numeric matrix is composed by `NumericValues`. Moreover, one `NumericValue` will either be a `NumericConstant` or `NumericVariable` over \mathbb{F} , or a gate emerging on NC which will eventually evaluate to a numeric value (e.g. $V_{\mathbb{F}}$). Likewise, new operator vertices on \mathbb{B} , like V_{\square} , will be encoded as `BooleanFormulas` whose inputs include, at least, one numeric value, meaning that even though they are technically implemented as a kind of Boolean structures, these cannot be used in a qualitative context (which is intended, as the need for such structures arises precisely from working on a quantitative perspective).

⁴ Even though in practice every `NumericValue` will also be a `BooleanValue` by inheritance, for the remainder of this dissertation, unless explicitly stated, "Boolean value" will always refer to an element of \mathbb{B} or a `BooleanValue` depending on the context, but never a `NumericValue`.


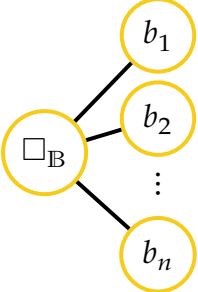
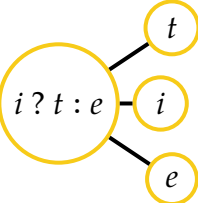
Class Name	Gate Structure	V	
NotGate		\neg Negation	Specifies the negation of the input BooleanValue b .
MultiGate		$\square_{\mathbb{B}} \in \{\wedge, \vee\}$ Logical AND and OR	Represents the result of the conjunction/disjunction of <i>at least</i> two Boolean inputs.
ITEGate		$\cdot ? \cdot \cdot \cdot$	If i turns out to be <i>true</i> then selects t , else e goes through.

Table 5.1: Boolean gates supported by Kodkod.

Unlike the CBC assembly process taken as reference, only a single factory will be used to build Numeric Circuits. However, a `NumericFactory` can be one of two kinds, depending on the type of problem at hand: a `NaturalFactory` ($\mathbb{F} = \mathbb{N}_0$) or a `StochasticFactory` ($\mathbb{F} = [0, 1]$). This factory is responsible for producing and managing numeric values (and Boolean values when necessary) and numeric matrices, that is, a `NaturalFactory` creates matrices from $Mat_{\mathbb{N}_0}$ and ensures that values like `NumericConstant` and `NumericVariable` belong to \mathbb{N}_0 , while `StochasticFactory` builds matrices from LS and guarantees that the numeric values correspond to probabilities.

For convenience and further flexibility, in actuality a `StochasticFactory` will be able to handle values over \mathbb{R} , supporting constraints which take advantage of such range (e.g. $1.5x + 2y > 3.5$). Nevertheless, the numeric values defining the relations that compose the model at hand will, in fact, be constrained to the $[0, 1]$ value range, meaning that potential instances which include values from outside that range are immediately discarded and, for example, if x corresponds to one of those numeric values, constraints like $x > 6.75$ immediately cause the specification to be deemed inconsistent.

As an extension of Boolean values, every `NumericValue` must have an *integer label* associated, which will still be taken advantage of during the NC assembly process and to apply the caching mechanisms already implemented for CBC (adapted to support this new kind of circuit) also. Despite the similar use

of labels, it is important to mention that these will not necessarily abide to the same properties as when constructing CBCs in the original Kodkod. Nevertheless, when building Numeric Circuits, the following characteristics are expected:

- Unlike Boolean constants, not all `NumericConstants` are shared between factories, as that is impossible due to the infinite domain of values being handled. Instead, only 0 and 1 are common, as they correspond to the only constants frequently used throughout this implementation, for both possible quantitative types of problems. Every other constant will be produced on demand by the factory responsible for that specific problem.
- The label of any vertex of the NC is unique up to its domain, Boolean or numeric. For instance, if two numeric values share the same label, they are the same numeric value, but if a numeric value and a Boolean value possess the same label, they do not depict the same vertex. However, given a label i , $b_i \in \mathbb{B}$ and $x_i \in \mathbb{F}$ are not the same but they must be related with one another precisely as described previously in Section 5.2, i.e., b_i corresponds to x_i perceived from the Boolean point-of-view and thus, $b_i = x_i > 0$ must always hold. For example, $x_1 = 3$ and $b_1 = false$ is trivially unsatisfiable for any Kodkod problem.
- Negative labels are only assigned to Boolean values which describe the negation of the nodes with the symmetric label, meaning that $b_{-i} = \neg b_i$ and/or $b_{-i} = \neg_{\mathbb{F}} x_i$.

In regards to `NumericVariables`, besides *labels*, they can also possess other kinds of characteristics associated, in order to easily enforce constraints as need be to fulfill this implementation, depending on the value's domain, scope, and so on:

- A finite list of potential numeric values, encompassing all the possible assignments that the tool is allowed to give this variable when finding an instance. If there is not at least one value from the list that can be attributed to it without violating other constraints, it means the problem has no solution in this circumstances.
- State explicitly that the variable must represent a *true* or a *false* value from the Boolean prospective, i.e., have a strictly positive value or be assigned 0.
- A `NumericConstant` indicating the maximum value that the variable will be able to assume.

All these properties are *optional* when specifying a variable, being simply required to belong to \mathbb{F} in case none of them are imposed.

FROM FOL TO NUMERIC CIRCUITS The Kodkod abstract syntax provides constructs to specify a **Kodkod problem** through first-order relational logic formulæ. Such problem is described by a *universe declaration*, enumerating the relations to be considered as well as their arity and typing, establishing the *bounds* of the problem, together with a first-order formula over the relations declared.

Thus the Kodkod abstract syntax was extended to make possible the description of quantitative constraints, as can be seen in Figure 5.5.

```

problem := univDecl relDecl* formula  expr := rel | var | unary | binary | comprehension
univDecl := { atom[, atom]* }         unary := unop expr
relDecl := rel :arity [constant, constant]  unop := ~ | ^ | drop
varDecl := var : expr                 binary := expr binop expr
                                        binop := + | & | - | . | -> | <: | >: | × | /
                                        comprehension := {varDecl | formula}

constant := {tuple*}
tuple := ⟨atom[, atom]*⟩

arity := 1 | 2 | 3 | 4 | ...
atom := identifier
rel := identifier
var := identifier

formula := elementary | composite | quantified | comparison
elementary := expr in expr | mult expr
mult := some | no | one
composite := not formula | formula logop formula
logop := and | or
quantified := quantifier varDecl | formula
quantifier := all | some
comparison := expr ineqop expr
ineqop := < | > | <= | >=

```

Figure 5.5: Extended Kodkod abstract syntax (adapted from (Torlak and Jackson, 2007)).

The original syntax remains unchanged, but now includes:

- The unary operator **drop** over an `Expression` to specify the Boolean correspondence of the operand expression;
- Domain/Range operators with respect to an `Expression` for a given `Expression` of arity 1. These operators already exist in the Alloy language, as seen in Section 2.6, but were not explicitly defined at Kodkod-level since within the Boolean context, they can be defined with the help of other elementary operators. However, it was deemed necessary to also explicitly add them to the Kodkod syntax, as a generalization of those operators for the quantitative context is not so easily achieved with the help of current operators only.
- Both the Hadamard product and division over two `Expressions` of the same arity.
- The comparison between two `Expressions` of the same arity using inequality operators, taking their weights into account.

It is important to note that, at the moment, `IntExpressions` and others of the same kind, i.e., the syntax used to manage integer values, on both Kodkod and Alloy, will keep the same naming pattern – as in **Int** –, however, their semantics will be generalized, so that the numeric values at hand:

- During the default Boolean analysis, they represent integers in two's complement;
- For quantitative analysis over natural numbers, they corresponds to integers within the $[0, +\infty[$ range;
- Finally, if it is intended to model probabilistic behaviour, then specification-wise, they work over real numbers (as mentioned previously, relations are required to be represented by a LS matrix and thus, certain constraints that go over the probabilistic range $[0, 1]$ will trigger trivially inconsistent models).

Having specified a Kodkod problem, the first step is to build the Numeric Circuit associated with the main formula. For that, a `LeafInterpreter` is used, which, as implied by its name, will be responsible for the **leaves** of the circuit, handling relations and constant expressions of the problem at hand, producing matrices composed by numeric values adequately to the **bounds** specified, as well as the analysis context \mathbb{F} in question. Therefore, after being handed the necessary problem data, this interpreter instantiates the suitable kind of numeric factory, which in turn is used produce the matrices taking the problem scope into account precisely as described previously in Section 5.3.

With the NC's leaves taken care of, the necessary gates and the edges between them and the leaf vertices must be determined to properly model the Kodkod problem considered. These are built by taking advantage of matrix operations, (re-)implemented to satisfy the requirements of matrices under \mathbb{F} . Most relational operators are implemented as the matrix operator deduced when studying the shift between the category *Rel* to *Mat*, as presented in Table 1.3:⁵

- Binary operators

COMPOSITION $R \cdot S$ is represented by matrix-matrix multiplication. While in the original Kodkod implementation, to ensure closure under Boolean matrices, \wedge is used instead of \times and \vee instead of $+$, between numeric matrices \times and $+$ are adopted once again.

RELATIONAL PRODUCT $R \rightarrow S$ is implemented as matrix cross product $R \times S$.

UNION $R \cup S$ will depend on the kind of matrices being subject to the operation, as mentioned in Section 2.2, in particular following the definitions established in Table 2.2. Between matrices of $Mat_{\mathbb{N}_0}$, union is defined as matrix addition $R + S$ (2.45). Under a probabilistic setting, the union of left-stochastic matrices corresponds to the normalization of matrix addition $norm(R + S)$ (2.47), with $norm$ being implemented precisely as defined in (2.48).

⁵ For simplicity, during the following descriptions $R[i]$ will be used to refer to the selection of the element i of the sparse-sequence storing the matrix representation of the relation R , i.e., $\llbracket R \rrbracket[i]$, where i is the flat-index of the matrix entry $[i_1, i_2, \dots, i_n]$, with n representing the arity of R .

INTERSECTION $R \cap S$ like union, depends on the type of matrices being dealt with and follows the definitions displayed in Table 2.2. It corresponds to the matrix containing at each cell the minimum value between the two elements in the same entry (2.44).

$$R = M \cap N \equiv \forall i. R[i] = \min(M[i], N[i])$$

Between left-stochastic matrices, additionally, the result is normalized (2.46) to ensure closure under LS .

DIFFERENCE $R - S$ is encoded as matrix subtraction, with the nuance that resulting negative matrix cells must be taken as 0-valued, to guarantee closure under numeric matrices over \mathbb{F} .

HADAMARD PRODUCT $R \times S$ between two relations corresponds to the Hadamard product between their respective matrices.

HADAMARD DIVISION R/S , likewise, is implemented analogously to the Hadamard product with $/$ instead of \times . To be noted that attempting division by zero on potential solutions at either translation level or during solving will be immediately deemed unsatisfiable. However, identifying these undesirable instances is not as easy as it may seem at first glance, due to the numeric matrix representation of each relation with respect to the problem’s bounds, described in Section 5.3.

For instance, consider the relations $Q, R, S : A \rightarrow B$ with $Q = R/S$ subject to the bounds $A \subseteq \{a_0, a_1, a_2\}$ and $B \subseteq \{b_0, b_1\}$. Then, these relations will be represented by the following matrices:⁶

$$R = \begin{matrix} & a_0 & a_1 & a_2 \\ b_0 & \begin{bmatrix} r_0 & r_2 & r_4 \end{bmatrix} \\ b_1 & \begin{bmatrix} r_1 & r_3 & r_5 \end{bmatrix} \end{matrix} \quad S = \begin{matrix} & a_0 & a_1 & a_2 \\ b_0 & \begin{bmatrix} s_0 & s_2 & s_4 \end{bmatrix} \\ b_1 & \begin{bmatrix} s_1 & s_3 & s_5 \end{bmatrix} \end{matrix} \quad Q = \begin{matrix} & a_0 & a_1 & a_2 \\ b_0 & \begin{bmatrix} q_0 & q_2 & q_4 \end{bmatrix} \\ b_1 & \begin{bmatrix} q_1 & q_3 & q_5 \end{bmatrix} \end{matrix}$$

r_i, s_i and q_i are NumericValues which represent the weight of the tuple associated, occurring within each relation respectively. Furthermore, since $Q = R/S$ then $q_i = r_i/s_i$ at circuit level. Thus, to avoid division by zero, one could simply require $s_i > 0$ for every $i \in [0, 5]$.

⁶ For clarity during this exemplification, only the relevant rows and columns are being considered. In practice, as mentioned before, at Kodkod level every dimension of every matrix is composed by the whole universe.

Now, by indiscriminately imposing such constraints, it will cause S (the denominator) to always meet the upper bound $A \times B$ and, consequently, $A = \{a_0, a_1, a_2\}$ and $B = \{b_0, b_1\}$ also, for every acceptable solution under these requirements. Yet, interpretations like:

$$\begin{aligned} A &= \{a_0, a_2\} & B &= \{b_1\} \\ R &= \{(a_0 \rightarrow b_1, 4), (a_2 \rightarrow b_1, 2)\} \\ S &= \{(a_0 \rightarrow b_1, 2), (a_2 \rightarrow b_1, 1)\} \\ Q &= \{(a_0 \rightarrow b_1, 2), (a_2 \rightarrow b_1, 2)\} \end{aligned}$$

should be an acceptable Alloy instance for the specification provided, as no division by zero occurs for the A, B, R and S provided, at this level. But, at Kodkod level, such solution is defined by the matrices:

$$R = \begin{matrix} & a_0 & a_1 & a_2 \\ b_0 & \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \\ b_1 & \begin{bmatrix} 4 & 0 & 2 \end{bmatrix} \end{matrix} \quad S = \begin{matrix} & a_0 & a_1 & a_2 \\ b_0 & \begin{bmatrix} 0 & 0 & 0 \end{bmatrix} \\ b_1 & \begin{bmatrix} 2 & 0 & 1 \end{bmatrix} \end{matrix} \quad Q = \begin{matrix} & a_0 & a_1 & a_2 \\ b_0 & \begin{bmatrix} \frac{0}{0} & \frac{0}{0} & \frac{0}{0} \end{bmatrix} \\ b_1 & \begin{bmatrix} 2 & \frac{0}{0} & 2 \end{bmatrix} \end{matrix}$$

$q_i = \frac{0}{0}$ for $i \in [0, 6] \setminus \{1, 5\}$, meaning that division by zero occurs on four divisions, and thus, this instance is discarded during solving. However, if the interpretation of a 0-weight tuple means that it does not occur within the relation for the instance at hand, then if a given tuple i does not occur either in the numerator nor the denominator ($r_i = s_i = 0$) then the division between the two should not be performed in the first place, and thus the tuple also does not appear in Q ($q_i = 0$).

Therefore, one could think of modifying the division constraints to $q_i = r_i > 0 \wedge s_i > 0 ? r_i/s_i : 0$ without enforcing $s_i > 0$. These constraints are still too strong, since if the numerator is a constant, i.e., $q_i = c/s_i$ with $c \in \mathbb{F}$ and $c > 0$ then a variant of the scenario initially considered occurs, $s_i > 0$ would be implicitly required, but instead, solutions where the tuple i does not occur in S ($s_i = 0$), it simply should also not occur in Q ($q_i = 0$). Changing the constraint to $q_i = s_i > 0 ? r_i/s_i : 0$ would also be too broad, as solutions where $r_i > 0$ and $s_i = 0$ that should be properly identified as unsatisfiable instances would be wrongfully accepted. More, all these scenarios are being considered over a division between two relations, but R and S can both be more complex relational expressions, making it all the more difficult to point out where division by zero is “acceptable” or not.

After studying the division between two general relational expressions R and S , the following mechanism to properly identify division by zero was implemented:

1. Every division is described by $s_i > 0 ? r_i/s_i : 0$.
2. With the Numeric Circuit associated with the given Kodkod problem assembled, a `DivisionDetector` will then traverse that circuit, gathering information over every division node that appears in it. In particular, for every division node n/d , identifies the set \mathcal{N} of all the primary variables that occur in the sub-nodes of n and the set \mathcal{D} of the primary variables composing the relational (sub-)expressions defined by the denominator d . That is, \mathcal{N} and \mathcal{D} specify the tuples that occur in the relational expression of the numerator R and the denominator S , respectively.
3. If division is used on this Kodkod problem, then another Numeric Circuit will be built, responsible for detecting divisions by zero as intended, as a V-gate, in general, whose inputs check if each division within the circuit attempts to divide by zero. That is, for every division node n/d , division by zero happens if either:
 - (a) $d = 0$ if $\mathcal{N} = \mathcal{D} = \emptyset$
 - (b) $d = 0 \wedge \bigvee_p^{\mathcal{N} \cup \mathcal{D}} p > 0$ otherwise
4. Finally, the *negation* of the circuit that detects division by zero is combined with the initial circuit, requiring instances that cause division by zero when the conditions are met to be properly deemed as unsatisfiable.

DOMAIN CONSTRAINT $V <: R$ contains the elements of R whose first dimension belongs to the vector V .

RANGE OPERATOR $R :> V$, similarly, results in a matrix with the entries of R whose last dimension occurs in V .

From the example in Figure 5.6 for binary relations, it can be easily observed that $<:$ filters column-wise while $:>$ filters by row. This implementation follows the line of thought pictured in the examples, generalized for relations of any arity $R : T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$ with $<:$ acting on T_1 and $:>$ on T_n , following the sparse-sequence structure and its flat-index mechanism.

Also, to be noted that the weights of V are meaningless under these operators, since there is only the need to check if a given element is 0-valued or not, that is:

$$(\text{drop } V) <: R \equiv V <: R \quad (5.3)$$

$$R :> (\text{drop } V) \equiv R :> V \quad (5.4)$$

- Unary operators

RELATIONAL CONVERSE R° corresponds to matrix transposition for both Boolean and numeric matrices (implementation remains unchanged).

$$\begin{aligned}
 \mathcal{A} &= A + B = \{a_0, a_1, a_2, b_0, b_1\} \\
 R : A &\rightarrow B \\
 R &= \{(a_0 \rightarrow b_1, w_0), (a_1 \rightarrow b_0, w_1), (a_2 \rightarrow b_0, w_2), (a_2 \rightarrow b_1, w_3)\} =
 \end{aligned}$$

$$\begin{array}{c}
 a_0 \\
 a_1 \\
 a_2 \\
 b_0 \\
 b_1
 \end{array}
 \begin{bmatrix}
 a_0 & a_1 & a_2 & b_0 & b_1 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & w_1 & w_2 & 0 & 0 \\
 w_0 & 0 & w_3 & 0 & 0
 \end{bmatrix}$$

$$\begin{aligned}
 \{a_0, a_1\} <: R &=
 \end{aligned}$$

$$\begin{array}{c}
 a_0 \\
 a_1 \\
 a_2 \\
 b_0 \\
 b_1
 \end{array}
 \begin{bmatrix}
 a_0 & a_1 & a_2 & b_0 & b_1 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & w_1 & w_2 & 0 & 0 \\
 w_0 & 0 & w_3 & 0 & 0
 \end{bmatrix}$$

$$\begin{aligned}
 R :> \{b_0\} &=
 \end{aligned}$$

$$\begin{array}{c}
 a_0 \\
 a_1 \\
 a_2 \\
 b_0 \\
 b_1
 \end{array}
 \begin{bmatrix}
 a_0 & a_1 & a_2 & b_0 & b_1 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & w_1 & w_2 & 0 & 0 \\
 w_0 & 0 & w_3 & 0 & 0
 \end{bmatrix}$$

$$\begin{aligned}
 &=
 \end{aligned}$$

$$\begin{array}{c}
 a_0 \\
 a_1 \\
 a_2 \\
 b_0 \\
 b_1
 \end{array}
 \begin{bmatrix}
 a_0 & a_1 & a_2 & b_0 & b_1 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & w_1 & 0 & 0 & 0 \\
 w_0 & 0 & 0 & 0 & 0
 \end{bmatrix}$$

$$\begin{aligned}
 &= \{(a_0 \rightarrow b_1, w_0), (a_1 \rightarrow b_0, w_1)\}
 \end{aligned}$$

$$\begin{aligned}
 &=
 \end{aligned}$$

$$\begin{array}{c}
 a_0 \\
 a_1 \\
 a_2 \\
 b_0 \\
 b_1
 \end{array}
 \begin{bmatrix}
 a_0 & a_1 & a_2 & b_0 & b_1 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 \\
 0 & w_1 & w_2 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0
 \end{bmatrix}$$

$$\begin{aligned}
 &= \{(a_1 \rightarrow b_0, w_1), (a_2 \rightarrow b_0, w_2)\}
 \end{aligned}$$

Figure 5.6: Application of the domain/range operator over a relation R described by its Kodkod matrix representation, within the universe A .

DROP The new operator `drop R` builds the Boolean matrix associated to the numeric matrix representing R , obtained by applying `drop` (presented in Section 5.2) to every element of R .

$$\forall i. (\text{drop } R)[i] = R[i] > 0 ? 1 : 0$$

- N-ary operators: N-ary relational union, intersection, and so on, are implemented as a generalization to their binary counterpart.
- `IfExpression` between two relations R_1, R_2 and a condition P results in another relation described by a matrix where each element corresponds to the same element of R_1 if the condition holds, and the entry of R_2 otherwise: $R = P ? R_1 : R_2$.
- Logical operators and other Boolean operators between formulas are implemented analogously to the original implementation.
- Comparison Formula

SUBSET $R \subseteq S$ is done by checking if the matrix $R - S$ is semi-negative, that is:

$$\bigwedge_i (R - S)[i] \leq 0$$

EQUALS $R = S$ is achieved in the obvious fashion, two relations are the same if their matrix representation coincides:

$$\bigwedge_i R[i] = S[i]$$

INEQUALITY The addition of the capability of comparing relations in Kodkod using inequality operators $R \sqsubseteq S$ is accomplished as follows:

$$\bigwedge_i R[i] \sqsubseteq S[i], \text{ with } \sqsubseteq \in \{<, >, \leq, \geq\}$$

- The **cardinality** of a relation R is obtained by the sum of its weights, i.e., the sum of its matrix entries:

$$\#R = \sum_i R[i]$$

- **MultiplicityFormulas** are implemented precisely as described in Section 5.2.

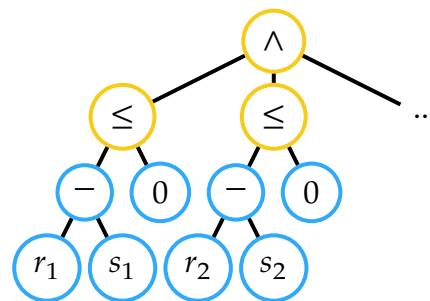
As mentioned in Section 5.4, when performing the usual Boolean analysis, integers are bounded, with each integer within the scope being explicitly represented by an atom of the universe \mathcal{A} , which allow for integer expressions to be characterized by the Boolean matrix representation (e.g. the before seen `Int` signature is the relation containing all the integers of the instance at hand). Under both quantitative settings, such explicit enumeration is impossible due to the infinity of the domains at hand, non-negative integers for $\mathbb{F} = \mathbb{N}_0$ and real numbers if $\mathbb{F} = [0, 1]$, but being able to handle numeric values by representing them through numeric matrices is still required. To accomplish that, this implementation takes advantage of the weights associated with each tuple, fixing an atom to represent numeric values in general, with the value itself being represented by the weight associated with it. For that, the usual integer representation is taken into account, using an element of `ints` (integers at Kodkod level) as the integer atom to represent the numeric values. Within this convention, the value of the integer atom fixed itself is meaningless, simply choosing the minimum integer value x within `ints` in practice, such that, $\{(x, w)\}$ will effectively represent $\{w\}$, with w corresponding to the value of the numeric expression, evaluated under \mathbb{F} . Like the Boolean counterpart, if no integer scope is specified, `ints` = $\{\}$ and thus, the matrix representation of any numeric value or expression will result in the null matrix (and consequently, the empty relation). Therefore, like will be seen later on, it must ensured at Alloy level that `Int` is non-empty to allow for easy manipulation of numeric values. However, this approach is unfortunately not enough to fully accommodate to the original semantics of integer's matrix representation, namely when subject to relational operators, since a single atom is being

used, currently only one integer may be referred at a time through this representation. For example, while $\{ 2 \} + \{ 3 \} = \{ 2, 3 \}$ originally, under quantitative scenarios $\{ 2 \} + \{ 3 \} = \{ 5 \}$, as every operation acts on the same atom. Thus, further work must be done in the future to fully be able to handle the infinite domains at hand through matrix representation using current structures while at the same time preserving the semantics of relational operations.

Now, if the Kodkod problem at hand is probabilistic, then the constraint that every relation must be represented by a left-stochastic matrix must be added to the formula at hand. Thus it was implemented a logical operator that returns *true* if the numeric matrix is left-stochastic and *false* otherwise: in case the relation is a set, it is determined if the vector associated represents a probabilistic distribution by simply checking if the sum of its elements add up to 1; if not, the matrix will be left-stochastic if $! \cdot \llbracket R \rrbracket = !$ (Oliveira, 2012a).

Above was highlighted the Numeric Circuits' assembly process from Kodkod for part of its abstract syntax. All the definitions described here and in previous sections are directly translated to the gates presented in Table 5.2 in a direct way: for operators like relational union, corresponding to matrix addition, it is simply the `ArithmeticGate` with $\square_{\mathbb{R}} = +$; while for other operators which are described by a more complex expression like subset, its definition is achieved by combining the gates supported:

$$\bigwedge_i (R - S)[i] \leq 0$$



where r_i and s_i represent $R[i]$ and $S[i]$, respectively.

Having assembled the Numeric Circuit associated with the problem at hand, if the first was reduced to a Boolean constant, then it means the problem is trivially (un)satisfiable and thus the solving process is done for this specific problem, otherwise the next step will be to generate the specification suitable to be fed into a quantitative solver (an SMT Solver or PRISM).

Class Name	Gate Structure	Operator(s)	
AritGate		Arithmetic operators $\square_{\mathbb{F}} \in \{+, -, *, /, mod\}$	Contains the result of applying the specified operator over all numbers received as inputs (at least two).
ChoiceGate		$\Delta \in \{min, max\}$ or $\cdot ? \cdot \cdot \cdot$	A binary gate responsible for choosing between two numeric values with respect to some criteria, assuming one of two forms: either chooses the minimum/maximum value between the two inputs; or given a predicate p , selects the first operand x_l if p holds and x_r otherwise.
UnaryGate		Unary operators $\diamond \in \{-, abs, sgn\}$	Applies the unary operator – negation, absolute value or sign – to the input number.
NumNotGate		\neg Numeric Negation	Negation of a numeric value from the Boolean point-of-view, i.e., represents $\neg(x > 0)$.
CmpGate		$\sqsubseteq \in \{=, <, >, \leq, \geq\}$	A Boolean binary gate, whose job is to specify the comparison between two numeric values under the (in)equality operator considered.

Table 5.2: Newly added gates in the quantitative extension.

5.6 SUMMARY

Throughout this chapter, the impact of the increased expressiveness from working under quantitative domains is studied, so that an approach between different alternatives arising from the new expressive power can be decided at both Kodkod and Alloy level.

It begins by establishing the correspondence between Boolean concepts and their quantitative counterpart, including relations, atoms and the operations offered by the tool originally. Then, the numeric structures adequate to support the information associated with the quantitative setting were derived, taking into account the existing Boolean ones and the requirements of this extension – numeric values of \mathbb{F} , matrices of numeric values, and Numeric Circuits, which emerge from the Compact Boolean Circuits.

Lastly, the assembly process between a Kodkod problem and the numeric structures considered is described.

QUANTITATIVE ALLOY

In this chapter, the definitions and properties settled in previous chapters will be put into practice to achieve the quantitative extension of both Kodkod and Alloy.

First, the quantitative extension of Kodkod is detailed, from being able to process quantitative problems described by numeric structures, as established in Chapter 5, in order to produce models suitable to the tools being taken advantage of to perform quantitative solving – SMT Solvers and PRISM – as well as their integration and management in Kodkod, to the interpretation of the outcome provided by those solvers into a quantitative Kodkod instance.

After achieving the quantitative extension of Kodkod, the bridge between the latter and Alloy will need to be built, so that Alloy will be able perform quantitative analysis, supporting quantitative Alloy specifications, their transformation into adequate Kodkod problems, the correspondence between a Kodkod instance and an Alloy solution established, as well as how the Analyzer's features conformed to the quantitative models produced.

Finally the quantitative analysis process coordinated to accomplish the system developed will be reviewed as a whole, subsequently presenting the resulting system architecture alongside with the tool's workflow when performing quantitative analysis.

The chapter ends with a brief overview on how to use the extension implemented and highlights some of its new capabilities.

6.1 KODKOD

To perform quantitative solving over Kodkod problems specified as described in Section 5.5, Kodkod will have its engine integrate and support the adequate solvers, in such a way that both the original qualitative solving and the new extension can function independently by request of the user.

FROM NUMERIC CIRCUITS TO SMT If the quantitative Kodkod problem is to be analysed for $\mathbb{F} = \mathbb{N}_0$ then an SMT Solver will be used and thus, an SMT-LIB specification must be constructed from the Numeric


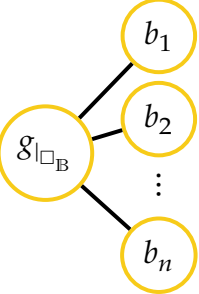
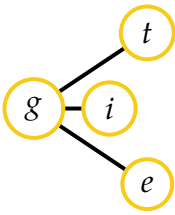
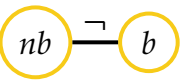
Node	SMT2-LIB	
	Operator(s)	Specification
BooleanVariable 		<code>(declare-const bi Bool)</code> <code>(= bi (> xi 0))</code>
MultiGate 	$\text{op} = \begin{cases} \text{and} & \text{if } \square_{\mathbb{B}} = \wedge \\ \text{or} & \text{otherwise} \end{cases}$	<code>(= g (op b1 b2 .. bn))</code>
ITEGate 		<code>(= g (ite i t e))</code>
NotGate 		<code>(= nb (not b))</code>

Table 6.1: Correspondence between the Numeric Structures and their respective SMT specification.

Circuit built. A `SMTGenerator` will be responsible for that task, pushing the constraints derived from the NC's components to the assertion stack. The correspondence between the circuit's leaves and gates, and the SMT specification is established in Table 6.1.

After deciding on the translation between the Numeric Circuit's components and SMT, it was possible to come to the conclusion that the specification should be verified with respect to the **theory of integers**, by taking the logic `QF_NIA` – *Quantifier-free integer arithmetic* – when solving, as it is the smallest fragment able to cover the kind of function symbols and assertions in the specification generated.

Furthermore the resulting SMT problem will possess certain characteristics:

- The function symbols of the specification generated will be exclusively of sort integer and Boolean, as they are used to represent the NC vertices, which can either be a numeric or Boolean value;


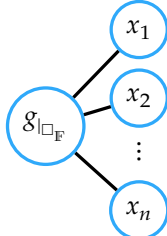
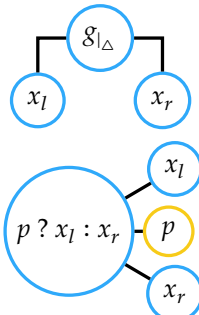
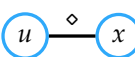

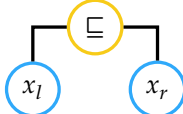
Node	SMT2-LIB	
	Operator(s)	Specification
NumericVariable 	$F = \begin{cases} \text{Int} & \text{if } F = \mathbb{N}_0 \\ \text{Real} & \text{if } F = [0, 1] \end{cases}$	<code>(declare-const xi F)</code>
AritGate 	$\text{op} = \square_F \text{ if } \square_F \in \{+, *\}$ $\square_F = -$	<code>(= g (op x1 x2 .. xn))</code> <code>(= g (ite (> (- x1 x2 .. xn) 0) (- x1 x2 .. xn) 0))</code>
	$\text{op} = \begin{cases} \text{div} & \text{if } \square_F = / \wedge F = \mathbb{N}_0 \\ / & \text{if } \square_F = / \wedge F = [0, 1] \\ \text{mod} & \text{if } \square_F = \text{mod} \wedge F = \mathbb{N}_0 \end{cases}$	<code>(= g (op (... (op x1 x2) ...) xn))</code>
ChoiceGate 	$c = \begin{cases} (< x_l x_r) & \text{if } \Delta = \text{min} \\ (> x_l x_r) & \text{if } \Delta = \text{max} \\ p & \text{otherwise} \end{cases}$	<code>(= g (ite c x_l x_r))</code>
UnaryGate 		<code>(= u (- x))</code> <code>(= u (abs x))</code> <code>(= u (ite (> x 0) 1 (ite (< x 0) -1 0)))</code>
NumNotGate 		<code>(= nx (not (> x 0)))</code>
CmpGate 	$\text{cmp} = \sqsubseteq$	<code>(cmp x_l x_r)</code>

Table 6.1: Correspondence between the Numeric Structures and their respective SMT specification (continued).

- Each function symbol has a unique identifier between function symbols of the same sort, which is exactly their label on the circuit;
- All function symbols follow a naming convention over their respective identifier, so that a Boolean and an integer function symbol with the same identifier must be related as specified in 5.2, as expected from the properties of the Numeric Circuit taken into account.
- Integers may or may not be bounded. Function symbols that arise from numeric variables are, at least, required to be non-negative. If an upper bound is imposed, then those function symbols will range between 0 and that limit. Intermediate integer function symbols, i.e., function symbols derived from numeric gates are always unbounded.

PROBABILISTIC SMT While the SMT Solver was initially intended to handle exclusively integer problems, as studied above, after developing this solution and checking which parts were tied to the analysis context itself, since it is also possible to work with SMT under the domain of real numbers, as seen in Section 3.1.1, it sparked the interest to check how this kind of tools would cope with the probabilistic setting and if it was, in fact, possible to handle such scenarios, how effective they would be.

As already illustrated in Table 6.1, only a few components of the Numeric Circuit depend on the \mathbb{F} , therefore `SMTGenerator` \leftarrow `ProbabilisticSMT` was implemented to generate the context-dependent specification adequately to $\mathbb{F} = [0, 1]$:

- Instead of integer function symbols, numeric values will give origin to **real** function symbols.
- The syntax for the division operator on `AritGate` must be adapted from Euclidean division to real number division. Therefore, the operator `mod` is not supported on this kind of analysis, as expected from the definition of Numeric Circuit.

During solving, the theory of integers is no longer suitable for this kind of specification, meaning that the **theory of reals** will be used instead, by adapting the logic fragment `QF_NRA` – *Quantifier-free real arithmetic* –.

Moreover, in the specification produced, real function symbols benefit from the same properties as their integer counterpart detailed above, with the difference that they are always bounded between $[0, 1]$, except for those arising from non-leaf vertices, which are still unbounded.

In the end, an object `SMTSpecification` is produced containing the following data:

`smt2` : `String` – the full SMT2-LIB specification generated.

`numFunctionSymbols` : `[Integer, String]` – the association between the numeric (`Int` or `Real`) function symbol's identifier and their denomination within the specification.


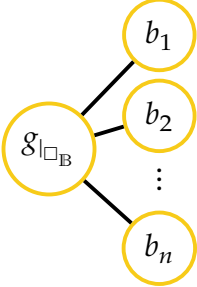
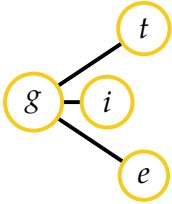
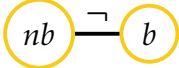
Node	PRISM	
	Operator(s)	Specification
BooleanVariable 		$b_i : \text{bool init false};$ $[] \text{ xi} > 0 \rightarrow (b_i' = \text{true});$
MultiGate 	$\text{op} = \begin{cases} \& & \text{if } \square_B = \wedge \\ & \text{otherwise} \end{cases}$	$[] b_1 \text{ op } b_2 \text{ op } \dots \text{ op } b_n \rightarrow (g' = \text{true});$
ITEGate 		$[] i \rightarrow (g' = t);$ $[] !i \rightarrow (g' = e);$
NotGate 		$[] b \rightarrow (nb' = \text{false});$ $[] !b \rightarrow (nb' = \text{true});$

Table 6.2: Translation of Numeric Structures into a PRISM model.

`numberOfVariables` : `int` – the total number of function symbols (both Boolean and numeric).

`numberOfAssertions` : `int` – size of the assertion stack.

`options` : `QuantitativeOptions` – solving options imposed by the user.

FROM NUMERIC CIRCUITS TO PRISM In case $\mathbb{F} = [0, 1]$, `Num2prismTranslator` shall transform the given circuit into a PRISM model.

The relation between the circuit's components and PRISM syntax is defined in the Table 6.2.

After translating the NC to the PRISM language, the various elements will be organized within a Discrete-Time Markov Chain, composed by a single module as pictured in Figure 6.1.


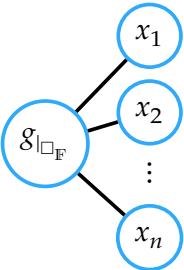
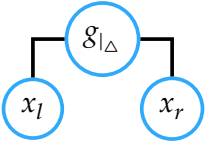
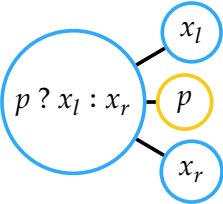
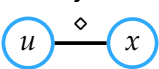
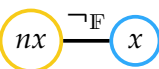
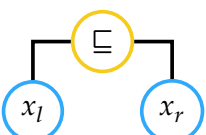
Node	PRISM	
	Operator(s)	Specification
NumericVariable 		<pre>const double xi;</pre>
AritGate 	$op = \square_{\mathbb{F}} \text{ if } \square_{\mathbb{F}} \in \{+, *, /\}$ $\square_{\mathbb{F}} = -$	<pre>formula g = x1 op x2 op .. op xn;</pre> <pre>formula g = x1 - x2 - .. - xn <= 0.0 ? 0.0 : x1 - x2 - .. - xn;</pre>
ChoiceGate  	$m = \begin{cases} \min & \text{if } \Delta = \min \\ \max & \text{if } \Delta = \max \end{cases}$	<pre>formula g = m(xl, xr);</pre> <pre>formula g = p ? xl : xr;</pre>
UnaryGate 	$\diamond = -$ $\diamond = abs$ $\diamond = sgn$	<pre>formula u = -x;</pre> <pre>formula u = x >= 0 ? x : -x;</pre> <pre>formula u = x > 0 ? 1 (x < 0 ? -1 : 0);</pre>
NumNotGate 		<pre>nx : bool init true;</pre> <pre>[] x > 0 -> (nx' = false);</pre>
CmpGate 	$cmp = \sqsubseteq$	<pre>b : bool init false;</pre> <pre>[] (xl cmp xr) -> (b' = true);</pre>

Table 6.2: Translation of Numeric Structures into a PRISM model (continued).

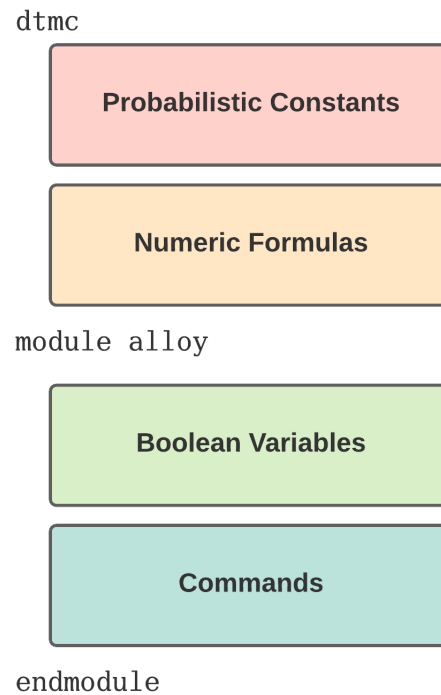


Figure 6.1: Generated PRISM model structure.

PRISM CONSTANTS represent the primary variables of the problem ($V_{\mathbb{F}}$), i.e., the numeric variables of the circuit derived from the problems' relations. These probabilities are represented in the model as *global double constants*.

Each variable is identified by a `PrismConstant` which characterizes the first by:

`label` : `int` – the unique identifier of this constant, coinciding with the label of the vertex that originated this constant.

`name` : `String` – its name in the model.

`minimum/maximum` : `Double` – the minimum/maximum value that this constant can assume (optional).

`range` : `Interval*` – represents the ordered set of intervals which specify the values that this constant can assume (optional).

This information will be essential during model checking using PRISM.

FORMULAS are the intermediate double expressions, arising from numeric gates.

VARIABLES correspond to Boolean PRISM variables within the module, identifying the Boolean values in the given circuit. They are of the form `b : bool init initialValue;`

COMMANDS of the generated model describe the conditions where the problem's (sub-)formulas and Boolean variables initial state change (either going from *false* to *true* or *true* to *false*):

Alongside the PRISM model, a PCTL formula is also produced in order to determine the satisfiability of the problem by model checking. Such formula takes the form

$$P \geq 1 \ [\ F \ \bigwedge_i f(i) \]$$

where f represents the top-level sub-formulas of the Kodkod problem in question. This is a qualitative property, meaning that the *true* response will be given for a valid solution of the problem, and *false* for an unsatisfiable potential instance instead. $P \geq 1$ requires the solution to hold with probability 100%, otherwise instances for which some of the core requirements of the model are not guaranteed to hold can be deemed as a SAT response.

Finally, a `PrismModel` is generated, described as follows:

```
mode : SolvingMode – intended mode to model check this model.1
prism : String – the full PRISM model.
props : String – property to be model checked.
constants : PrismConstant* – primary variables of this model.
numberOfFormulas : int – number of formulas in this model.
numberOfCommands : int – number of commands within the module.
options : QuantitativeOptions – solving options.
```

QUANTITATIVE SOLVERS After deriving either an SMT specification or a PRISM model, the next step is to pass it to the respective solver in order to reach a conclusion about the satisfiability of the Kodkod problem at hand. For that, at least one SMT solver and PRISM must be integrated into Kodkod.

First, to better manage the life cycle of any kind of solver in the quantitative context, an interface `QuantitativeSolver` was defined, containing all the necessary methods to interact with a quantitative solver, independently of the context, that is, both common to an SMT Solver and PRISM, some of which can be highlighted:

```
solve : boolean – Checks the satisfiability of the model at hand, returning true iff it is satisfiable,
and false otherwise, meaning that it is unsatisfiable if PRISM was used, but either the response
UNKNOWN/UNSAT was obtained if it was solved with an SMT Solver;
```

¹ `SolvingMode` will be seen in more detail later on this section.

`getValue(label : int) : Number` – Given a SAT response, returns either an integer or a double (probability), depending on \mathbb{F} , associated to the numeric variable with that label in the instance found by the solver;

`getBooleanValue(label : int) : boolean` – Analogously to `getValue`, if the problem is satisfiable, returns the Boolean value determined in the most recent instance for the variable with the given label associated. This method abides to the relationship between Boolean and numeric values established previously in Section 5.2, meaning that even if there was not explicitly any Boolean variable/function symbol in the PRISM model/SMT specification with that identifier, as long as there is a numeric value x with such label, this method will return $x > 0$ in this scenario. This is useful, for example, to check if a certain tuple occurs in its relation for that particular instance.

`elimSolution(labels : Integer*)` – Forces the solver to ignore solutions where the primary variables identified by the given labels are assigned the same values as the newest solution found.

SMT SOLVER While in this implementation CVC4 is the only SMT Solver integrated, in a first attempt by taking advantage of its own JAVA API, to ease the integration of other SMT Solvers in the future, a more flexible approach was taken, by managing an instance of the solver through its binary. So, the more specific `QuantitativeSolver` \leftarrow `SMTSolver` interface was defined, with further methods to manage SMT Solvers in general. In particular, the method `getResult : SMTResult` is required, so that the precise judgement SAT/UNKNOWN/UNSAT can be extracted after the method `solve` is called.

Furthermore, as long as the SMT Solver follows the SMT2-LIB standard, a very similar implementation of the `CVC4Solver` is enough to integrate other solvers. For instance, the `solve` workflow can be roughly depicted as follows:

```

1: function solve( )
2:   solver.write(smt2)
3:   solver.write("(check-sat)")
4:   result ← parseResult(solver.readLine())
5:   if result == SAT then
6:     // parse instance
7:     solver.write("(get-model)")
8:     while (line ← solver.readLine()) != null do
9:       solution.add(parseSolution(line))
10:    end while
11:  end if
12:  // else result == UNSAT || UNKNOWN
13:  return result
14: end function

```

where `solver` is the instance of the SMT Solver; `smt2` is the full specification; `write` sends commands to the solver and `readLine` obtains the response; with `parseResult` and `parseSolution` being the main solver dependent methods, which should be capable of consuming the response data in the format output by the solver, meaning that the implementation must conform to the properties of the specification produced by the `SMTGenerator`, and thus only SMT problems which abide to those characteristics can be fed into the solver, otherwise the parsing methods and the storing of the solution may behave in unexpected ways.

PRISM Similarly to CVC4, PRISM will be accessed through its binary, and the interaction with it is achieved using its command-line interface. As mentioned previously when describing the generation of a `PrismModel`, the latter will be verified according to a `SolvingMode`, one of the various ways PRISM offers to analyse the model at hand. Currently, the following are implemented:

- `SINGLE_QUALITATIVE` model checks the PRISM model with respect to a **qualitative** property by creating a single model checking instance at a time;
- `SINGLE_QUANTITATIVE` is similar to its qualitative version, with the only difference being that a **quantitative** property will be taken into account instead;
- `EXPERIMENT` creates a model checking instance for every possible solution to this model, with respect to a *step*, like was seen previously in Section 4.1.3, for a qualitative property;
- `PMC` performs parametric model checking over this model with respect to a qualitative property.

A `PRISMResult` is responsible for storing the solution obtained depending on the solving mode considered. Before solving, the PRISM model and the property to be checked are written into `alloy.prism` and `alloy.props` files, respectively. Afterwards, depending on the solving mode and taking into account the properties of the model's `PrismConstants`:

- `SINGLE_QUALITATIVE / SINGLE_QUANTITATIVE`

For every constant `c1`, `c2`, ..., `cn`, randomly selects a potential solution (one which abides to all the possible minimum/maximum/range constraints identified in the respective `PrismConstant` instance) `v1`, `v2`, ..., `vn`, which is then model checked through the following command:

```
$ prism alloy.prism alloy.props -const c1=v1, c2=v2, ..., cn=vn
  --exportresults outputfile
```

If `true` / probability higher than 0 is written to the output file, then the problem is satisfiable for that solution, otherwise another random solution (never equal to already checked solutions) will be tested, continuing the cycle until a satisfiable solution appears or once there are no other possible solutions abiding to the constraints.

- `EXPERIMENT`

Creates a PRISM experiment ranging the constants over the interval `[minimum, maximum]` (in case they are not specified in the respective `PrismConstant` instance, since they must be valid probabilities, defaults `minimum` to 0 and `maximum` to 1) considering a certain step as follows:

```
$ prism alloy.prism alloy.props
  -const c1=min1:step:max1, c2=min2:step:max2,
  ..., cn=minn:step:maxn
  --exportresults outputfile
```

Then, `PRISMResult` will associate to every solution checked the result (UN)SAT obtained.

- `PMC`

Executes parametric model checking taking the constants' range into account and assuming the given step as the precision.

```
$ prism alloy.prism alloy.props
  -param c1=min1:max1, c2=min2:max2,
  ..., cn=minn:maxn
  -paramprecision step --exportresults outputfile
```

The result obtained will then be described by a general rational function over the parameters, with regions of the last mapped to truth values, or by intervals of values for each constant together with the satisfiability result, similar to the example presented in Table 4.3.

It should be noted that the `step` $\in [0, 1]$ is specified by the user. Furthermore, `SolvingMode` would ideally also be chosen by the user, but given the current implementation, for reasons which will be discussed in greater detail in the last chapter, `SINGLE_QUALITATIVE` is the mode currently in use.

QUANTITATIVE INSTANCE Finally, after going from the original Kodkod problem into an adequate specification for the selected solver, a `QuantitativeTranslation` (an extension of `Translation` for qualitative analysis) is responsible for storing the problem details, the translation and the `QuantitativeSolver` instantiated to handle the specification generated, being one of two types: a `SmtTranslation` contains the SMT problem produced as well as the instance of the SMT Solver, while a `PrismTranslation` manages the PRISM model.

Once the model is solved, if a SAT response is obtained, `QuantitativeTranslation` offers the method `interpret` which is responsible for mapping the relations into sets of weighted tuples based on the most recent model determined by the solver. In order to create such `Instance`, a `TupleFactory` is needed to produce the tuples and tuple sets from the universe of the Kodkod problem in question.

However, since in a quantitative context each relation has a weight associated to each tuple, a regular `TupleSet` is not enough to represent a quantitative instance. Therefore `TupleSet` \leftarrow `QtTupleSet` was created, being able to store the weight for each of its tuples. Furthermore `TupleFactory` was augmented with the creation methods for this new kind of tuple set.

With that, `interpret` takes advantage of the `QuantitativeSolver` interface to extract the weight associated to each tuple with respect to the instance determined, producing `QtTupleSets` accordingly. An example of a relation from a quantitative instance is as follows:

$$R = \{ (a \rightarrow b_0, 1), (a \rightarrow b_1, 3) \}$$

To be noted that if a relation is represented by a Boolean matrix for a given instance, that is, every tuple present has weight 1, then `TupleSets` are used instead.

Having produced a possible instance of the Kodkod problem, one can interact with it and further analyse it through the use of an **Evaluator**, which offers the methods:

- `evaluate(Formula) : boolean`
- `evaluate(Expression) : TupleSet`
- `evaluate(IntExpression) : Number` (originally of type `int` instead, but due to the nature of values being handled in the quantitative context, to prevent rounding, generalization was needed)

which are able to determine the value of a relational formula or expression or an `IntExpression` (once again, `Int` for the qualitative context, but \mathbb{F} under the quantitative setting) over a specific instance. Therefore it was needed to implement a `QTEvaluator` to meet the same functions over a quantitative instance.

The evaluation process implemented is inspired by the existing qualitative version: the `Formula/Expression/IntExpression` is transformed into a `Numeric Circuit` and, since a solved instance is taken into account, the `LeafInterpreter` is able to assign constant values to the circuit's leaves, therefore during the assembly process every gate can be immediately simplified. In the end, the circuit will be reduced into a single vertex when evaluating formulas and `IntExpressions`, a `BooleanConstant` and `NumericConstant` respectively; evaluating a relational expression will result in a circuit with multiple `NumericConstant` nodes but no edges, describing the elements of the resulting matrix to be interpreted into a `QtTupleSet` (or a `TupleSet` if it is a Boolean matrix).

This implementation of the Kodkod evaluator, arriving at the response at translation-level is one of the reasons why the tuples described through comprehension had to default to weight 1, as mentioned in Section 5.2, since if other values were checked, then the resulting circuit wouldn't be constant and there would be the need to use the solver to find the response. Moreover, that also means that the same instance would potentially possess multiple solutions for the same comprehension expression, thus, handling this kind of expression thoroughly would require finding a general form describing all the possible solutions for a given instance, which was deemed out of the scope for this tool and project.

Ultimately, quantitative analysis of Kodkod problems was accomplished. In order to take advantage of the new capabilities first, the Kodkod problem must be defined: both the main `Formula` and the universe `Bounds` are specified exactly the same as to perform qualitative analysis (with the difference that within `Formula` may occur the use of the new additions to the Kodkod abstract syntax); solving `QuantitativeOptions` must also be provided, which include:

`analysisType` : `QuantitativeType` – represents \mathbb{F} , that is, the solver will work under the `INTEGER` or `PROBABILISTIC` context.

`solver` : `QuantitativeSolver` – the solver to be used during analysis. Currently supported:

- `CVC4API` for `INTEGER` (the implementation using the `CVC4 JAVA API`);
- `PRISM` for `PROBABILISTIC`;
- `CVC4` supports both `INTEGER` and `PROBABILISTIC` solving.

`maximumWeight` : `Integer` – used under \mathbb{N}_0 , specifies the maximum weight that each tuple may assume.

`incremental` : `boolean` – true iff `this.solver` will perform incremental solving.

step : `double` – relevant when using PRISM, as seen previously for experiments and parametric model checking.

Afterwards, the problem can be solved using either of two methods added to the `Solver` (KodkodSolver API):

- `solve`, which goes through the whole process described, passing the SAT/UNSAT (or UNKNOWN if an SMT Solver is used) judgement, and determining a single quantitative instance in case the problem is satisfiable;
- `solveAll`, which is similar to `solve`, but also offers an iterator over various solutions, allowing the user to surf over multiple quantitative instances until the UNKNOWN/UNSAT response is obtained. To achieve this, it was needed to develop a `QTSolutionIterator`, inspired by the `SolutionIterator` for SAT Solvers, but for quantitative solvers instead.

After solving, these methods also register solving statistics like their qualitative counterpart. So `SMTStatistics` and `PrismStatistics` were implemented as extensions of the existing `Statistics` to be able to report their solver's data properly. Lastly an object `Solution` will be produced, containing the `Outcome` (SAT, UNSAT or UNKNOWN), quantitative `Instance` (null if `Outcome` ≠ SAT) and the solving `Statistics`.

6.2 ALLOY AND THE ALLOY ANALYZER

With the quantitative extension of Kodkod up and running, the final phase of development is to adjust the Alloy components to be able to handle the new Kodkod and allow the user to perform quantitative analysis on demand.

Initially the Alloy **Core** must be extended in order to take advantage of Kodkod, which includes extending its own language to support the new quantitative operators, modify the correspondence between an Alloy specification and the respective Kodkod problem to define the latter properly under a quantitative setting, solve the problem derived using the quantitative methods provided by the Kodkod extension and then be able to interpret the eventual quantitative instances produced and properly establish the quantitative Alloy instance associated.

Afterwards, the Alloy **Application** – the Alloy Analyzer – should be able to support the new kinds of analysis, allowing the user to freely specify the setting for which the Alloy specification must be subject to during solving alongside which solver to use, being also able to report the results obtained accordingly. Furthermore both the Alloy Visualizer and Evaluator will need to adapt to the quantitative Alloy instances produced and faithfully present the outcome to the modeler.

LANGUAGE To fully accommodate to the new modeling capabilities of the extended Kodkod, allowing the user to specify further quantitative constraints, the Alloy language had to be modified as follows:

- `drop` was added to the language keywords and the Alloy parser was adjusted to interpret *drop* as an unary operator over relational expressions;
- the inequality operators between Alloy integers (`<`, `>`, `<=`, `>=`, `!<`, `!>`, `!<=`, `!>=`) can now be also used between two relational expressions of the same arity;
- to encode both Hadamard product and division, the built-in Alloy functions `fun/mul` and `fun/div` over two integer expressions, representing integer multiplication and division, are overloaded to specify the Hadamard operators when applied between two relational expressions with coinciding arity.

ALLOY TO KODKOD Defining the transition from a given quantitative Alloy specification into the adequate Kodkod problem is somewhat smooth due to how similar it is compared to the qualitative implementation, as the problem's bounds are derived from the Alloy scope identically for every analysis context, plus the correspondence between the Alloy constraints and the resulting Kodkod formula possesses only a few differences between the two kinds of domains supported, which is expected, as this implementation has its basis on the *scalable modeling* lemma, the same Kodkod problem can be tackled either with qualitative or quantitative semantics.

However, since the tool was also enriched with capabilities which are only useful in the quantitative setting, `TranslateAlloyToKodkod` \leftarrow `TranslateQTAAlloyToKodkod` was developed to handle those cases, as well as other variations, with the translation process coinciding with `TranslateAlloyToKodkod` save for the following:

- the `drop` of an Alloy expression corresponds to the `drop` of the respective Kodkod expression. To be noted that, since `drop` was added to the Alloy language as a whole, even specifications to be solved under the Boolean domain can technically contain the use of `drop`, although in such environment it has no effect, since it behaves as the identity function $drop\ R = id\ R$, meaning that an Alloy specification where `drop` occurs and the same specification with every use of `drop` removed will give origin to the exact same Kodkod problem when performing qualitative analysis.
- unlike the qualitative version, the Alloy *domain* and *range* operators will be encoded explicitly through the new Kodkod domain and range operators.
- on a quantitative domain, the use of inequality operators now needs to be distinguished between the comparison of two numeric values or the comparison between two relational expressions, remaining the same as the qualitative version for the first case, and encoding the comparison of the Kodkod expressions associated for the second, supported by the extended abstract syntax of Kodkod.

- the use of `fun/mul` and `fun/div` between two Alloy integers is represented by the multiplication and division of Kodkod integers. The line of thought remains the same when applied over (generalized) numeric values on a quantitative setting. In this scenario, there is also the need to check if the operands are now relations, in which case the corresponding Kodkod expressions will be subject to the Hadamard product and division. For example,

$$\begin{aligned} \text{Alloy} &\rightarrow \text{Kodkod} \\ 2 \text{ fun/mul } 6 &\rightarrow 2 * 6 \\ R \text{ fun/mul } S &\rightarrow R \times S \end{aligned}$$

- It was hinted at, in Section 5.1, that the Kodkod constraints associated with the declaration of Alloy *signatures* and *fields* would need to be carefully handled in the quantitative context. This is especially true once the multiplicity keywords are added to the mix.

For *signatures*, declarations like `(l)one sig S{...}` must explicitly enforce the `(l)one` constraint to the Kodkod expression associated to `S`, unlike during qualitative analysis, in which case the Kodkod problem can be optimized and not need to have such constraint defined explicitly, but enforced through the problem's bounds instead – which is not enough in a quantitative setting.

In regards to *fields*, within a quantitative context, for a n -ary relation $R : T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$, generally specified as

```
m1 sig T1{ R : T2 m2 -> m3 T3 m4 -> m5 T4 m6 -> .. -> m(2(n-2)+1) Tn }
```

the following Kodkod constraints must be imposed:

RELATIONAL TYPE First and foremost, $\text{drop } R \subseteq T_1 \times T_2 \times \dots \times T_n$ is required, in contrast to simply $R \subseteq T_1 \times T_2 \times \dots \times T_n$ in the qualitative version. This is necessary due to the fact that the latter constraint will result in R being declared as an ordinary relation represented by a Boolean matrix, as T_1, T_2, \dots, T_n are all composed by atoms and thus, their weights never exceed one, meaning that the product between them also never will, upper bounding R 's weights at a maximum value 1. By using *drop* on R , it is perceived as its Boolean representation, viewing any non-zero weight as 1 and thus, modeling the proper bounding constraint, as intended. For relations declared without using multiplicity keywords (omitted `m1 = m2 = .. = set`), the definition presented would be enough, but since in practice that is not the case, the constraints generated take a different form, however following the same train of thought.

MULTIPLICITY CONSTRAINTS To ensure that the relation is properly bounded with respect to its type, while also abiding to the multiplicity declarations, the constraints are imposed column-to-column, meaning that a similar change to the one proposed before has to be made to the corresponding

formula, applying *drop* carefully to make sure that only instances where the relation respects the multiplicity adequately are considered.

The first column, the *signature*, already has its multiplicity constraints handled as mentioned previously, being common to every field declared inside the first. Now, to make sure that the first column of the relation itself is well-bounded, the constraint established above is written with respect to the parent *signature*, i.e., $A \cdot (..(A \cdot (A \cdot R))) \subseteq T_1$ during qualitative analysis and now $drop(A \cdot (..(A \cdot (A \cdot R)))) \subseteq T_1$ when performing quantitative analysis, where A is the *signature* relation containing all the atoms in the universe at hand.

Finally, the remaining columns are handled in a nested fashion, by universally quantifying the column at hand consecutively, imposing the bound with respect to the signature associated with the column similarly to what was shown for the first column, alongside the multiplicity constraints for (1) one, if applicable (set does not require further constraints). Transitioning from the qualitative constraints to a quantitative scenario by applying *drop* to every (sub-)expression where R occurs indiscriminately analogously to the constraint considered for the first column is tricky, since it might change the meaning of the intended expression.

Using *drop* over the constraint bounding the column in focus not only preserves its intention, but is also necessary to do so, as without *drop*, this expression will cause the relation to behave like its Boolean counterpart, since the right-hand side operand of the subset expression is a *signature*, whose weights do not exceed the value 1, which is solved by applying *drop*, as already discussed previously for similar scenarios.

However, *drop* cannot be applied in the same manner on the sub-expressions that are delimiting the relation's columns to the adequate multiplicities, namely within the (1) one constraints, otherwise imposing properties like “one M(drop R, c, c')” – where $M(S, c, c')$ is the expression that will be subject to the multiplicity constraint, which describes the column c of the relation S with respect to the column c' , representing either the previous or the next, i.e., given “.. b -> l c r -> d ..” then “l M(S, c, b)” and “r M(S, c, d)” will be required to hold, if $l, r \in \{ \text{1one, one} \}$ – will incorrectly specify the multiplicity, since it will hold as long as that column contains a single tuple with weight **higher** or equal to 1, due to the constraint being specified over R from the Boolean point-of-view, which would violate the definition of the (1) one constraints imposed previously in Section 5.2. Furthermore, the need to take advantage of *drop* that arose from the multiplicity set is not there for these multiplicity keywords.

Taking everything into account, the process of generating the Kodkod constraints associated with the relation declaration is split into two steps:

1. Translating from Alloy in the same way as in the qualitative setting, while applying *drop* to **every** occurrence of the relation at hand.
2. Traverse the resulting Kodkod formula, removing every use of *drop* which violates the constraint purpose, that is, every occurrence of *drop* within a (\perp) one expression. This was achieved by implementing a traversal over the Kodkod AST, responsible for detecting in which cases *drop* was being misused and properly adjusting the formula afterwards.

For example, a relation declaration `sig A { R : B lone -> one C }` is described by the following Kodkod constraints, depending on the analysis context:

- Boolean analysis

$$\begin{aligned}
 & \forall a \in A. (((R \cdot \underline{a} \subseteq B \times C) \wedge \forall b \in B. \text{one}((R \cdot \underline{a}) \cdot \underline{b}) \wedge (R \cdot \underline{a}) \cdot \underline{b} \subseteq C) \\
 & \quad \wedge \\
 & \quad \forall c \in C. \text{lone}(\underline{c} \cdot (R \cdot \underline{a})) \wedge (\underline{c} \cdot (R \cdot \underline{a})) \subseteq B) \\
 & \quad \wedge \\
 & (A \cdot (A \cdot R)) \subseteq A
 \end{aligned}$$

- Quantitative solving

$$\begin{aligned}
 & \forall a \in A. (((\text{drop } (R \cdot \underline{a}) \subseteq B \times C) \wedge \forall b \in B. \text{one}((R \cdot \underline{a}) \cdot \underline{b}) \wedge \text{drop } (R \cdot \underline{a}) \cdot \underline{b} \subseteq C) \\
 & \quad \wedge \\
 & \quad \forall c \in C. \text{lone}(\underline{c} \cdot (R \cdot \underline{a})) \wedge (\underline{c} \cdot \text{drop } (R \cdot \underline{a})) \subseteq B) \\
 & \quad \wedge \\
 & \text{drop } (A \cdot (A \cdot R)) \subseteq A
 \end{aligned}$$

When defining the Kodkod problem's bounds from the Alloy scope, the only difference lies within the integer representation. To fulfill the relational representation of a numeric expression, as imposed in Section 5.5, non-empty integer bounds are enforced. Furthermore, a single integer atom is enough to fully handle numeric values from the whole domain, however one expression at a time. Therefore, no matter the bitwidth provided, `Int = { 1 }` will be set in Alloy (and consequently, `ints = { 1 }` in Kodkod) during quantitative analysis, as there is no reason to bound the numeric values' domain by default nor it is useful in the current approach to add further integer atoms to the universe, as they will not be used in practice and will only increase the problem size. Given its usage within Kodkod, numeric values represented by a numeric matrix, for example the value 0.75 will just be cast to $\{(1, 0.75)\}$. As stated previously, the value of the integer atom itself, 1 in the current implementation, has no meaning, with the value of the numeric

expression on \mathbb{F} being simply displayed as $\{0.75\}$ to the user in practice. Like already mentioned at Kodkod level, the representation implemented does not fully support the relational representation intended, as no more than one numeric expression can be represented in the same matrix at a time. In practice, the `ScopeComputer` responsible for handling the Alloy scope, other than defining `Int` accordingly depending on the type of analysis, follows the same process for the remaining components of the model at hand independently of the analysis context.

In the end, the quantitative Kodkod problem associated with a given Alloy specification was successfully derived.

QUANTITATIVE ALLOY INSTANCE An instance of `A4Solution` is responsible for storing the solution to a given problem, while also acting as the interface between Alloy and Kodkod. Thus, to accommodate to the quantitative version of Kodkod, it was enhanced with the following capabilities:

SOLVER A `solve` method suitable for quantitative solving was introduced, whose workflow is adapted from the qualitative version, but taking advantage of the Kodkod extension instead. After obtaining the Kodkod problem at hand from `TranslateQTAloyToKodkod`, its `Formula` and `Bounds` are specified, and ready to be fed into Kodkod. The next step before solving is to handle the solving options preferred by the user, that is, the `QuantitativeOptions` desired. Alloy's solving configurations are stored in a `A4Options` object and thus, the latter also had to be extended to include quantitative specific details, namely the analysis context at hand – \mathbb{B} or \mathbb{F} –, and the quantitative solver selected, if applicable. As mentioned previously in Section 5.4, during integer analysis, the integer scope associated with the Alloy command at hand will be used to limit the weight that a given tuple is allowed to assume. Therefore, taking all the options into account, the `QuantitativeOptions` object is built, storing the information about the value of \mathbb{F} considered, which solver will be used, whether the tuple's maximum weight is unlimited or upper bounded by a specific natural numbered value, and so on.

With this, all the conditions to perform quantitative solving have been met, so, either the method `solve` or `solveAll` of Kodkod (seen previously in Section 6.1), depending on the selected solver being able to do incremental solving or not, will be called. Afterwards, `A4Solution` also stores the outcome and, if the problem was satisfiable, the `Instance` produced, as well as an instance of the `Quantitative Evaluator` over that model.

When using an incremental solver, by interacting with this solution, the next instance can be requested and, by taking advantage of the solution enumerator, a new `A4Solution` is produced, coinciding with the original one, except that it contains the following outcome produced by the iterator, and the new instance in case it was found.

EVALUATOR If the `A4Solution` object has a satisfiable instance, then it is possible to interact with it by evaluating Alloy expressions with respect to the first. For that, it was necessary to extend `A4Solution` to provide evaluation methods suitable for quantitative instances, in which case the Alloy expression specified is translated into its proper Kodkod correspondence by `TranslateQTAlloyToKodkod`, which in turn is passed as argument to the quantitative Kodkod evaluator, already instantiated over the solution obtained after solving, as mentioned previously.

Now, once a Kodkod instance or an output from the Evaluator is obtained, these results must be transformed in the respective Alloy representation. Turns out that it is a rather smooth transition, as they are characterized through identical structures in both Kodkod and Alloy, in particular, a Kodkod `Tuple` corresponds to an Alloy `A4Tuple` and a Kodkod `TupleSet` is equivalent to `A4TupleSet` in Alloy. In a quantitative Kodkod instance, a `QtTupleSet` is used, storing the weights associated with the tuples also. Adjusting the Alloy representation to support weights can be done in a similar fashion, by adding `A4Tuple` \leftarrow `A4QtTuple`, which contains the weight associated with the given tuple. While in the quantitative extension of Kodkod, the weights were added at set-level, in Alloy it is handled at tuple-level. It is worth noting that both representations are equivalent, and this implementation decision was simply made due to convenience reasons during development, managing weights within the set in Kodkod was practical, but manipulating them at tuple-level in Alloy allowed for a cleaner implementation of the extension, compared to the set-level representation if it was used instead.

An object `A4Solution` can be written into a XML file or built from one, and for that, Alloy provides the helper classes `A4SolutionWriter` and `A4SolutionReader`. These are core components of the workflow within the main Alloy application, as the Alloy Analyzer manages the models found for a given specification through the `A4Solution` characterized by its XML representation using them. Therefore it is necessary to include in the XML files produced, the data necessary to handle quantitative instances and thus, these classes had to be extended as follows:

- `A4SolutionWriter` specifies within the `instance` node an attribute `context` whose value can either be “Boolean”, “Integer” or “Probabilistic” specifying the kind of analysis that the model at hand was subject to.

Moreover, it is also now responsible for encoding the weights associated to each tuple within the quantitative instance being written. In particular, if the `A4Tuple` to be written into XML is an `A4QtTuple`, then inside the `tuple` node is added the element `weight` containing the attribute value.

- Since the XML file now contains new information, the reader also had to be adjusted to properly extract it: the `context` is stored in the `A4Options` associated with this solution; when parsing the

Instance data, alongside the other tuple information, the weight is also stored, forming the proper `QtTupleSet` associated with each relation.

ALLOY ANALYZER At this point, all there is left to do is to adjust the Alloy Analyzer and its features to take advantage of the quantitative extension.

`SimpleGUI` is responsible for managing the graphical interface and the application's functionalities, therefore being the main focus where the needed modifications will be made.

First, `A4Preferences`, which contains the customizable options given to the user, will be extended to present the following alternatives:

ANALYSIS CONTEXT The choice between “Boolean”, the default qualitative analysis, “Integer” or “Probabilistic” solving;

SMT SOLVER The SMT Solvers available for “Integer” analysis, which currently corresponds to CVC4 only (“CVC4” and “CVC4API” are given as options, being the two integrations mentioned in Section 6.1, where the first executes the CVC4 binary and the second uses the CVC4 JAVA API, effectively performing the same);

PROBABILISTIC SOLVER Analogously, when “Probabilistic” analysis is intended, selects which solver will be used, either being “Prism” or “CVC4”.

These are added to the `Options` menu in the graphical interface to be freely adjusted by the user between solving attempts.

Once the user intends to start the automatic solving, by pressing *Execute*, the `SimpleGUI` will be responsible to create the `A4Options` containing the solving options selected, including the newly added quantitative analysis information; then if the context selected is “Boolean”, it will launch the `SimpleTask1`, which is responsible for solving the given command(s) for the first time, proceeding exactly like the original Alloy; otherwise, a `QuantitativeTask` is executed, responsible for handling the model at hand analogously to `SimpleTask1`, but for either “Integer” or “Probabilistic” problems instead. Furthermore, if the problem is satisfiable and the modeler desires to find further solutions, then a `SimpleTask2` will be created to handle the enumeration in the “Boolean” setting, while the `QTEnumerationTask` will be responsible for the solution iteration within a quantitative context.

QUANTITATIVE TASK Both `QuantitativeTask` and `QTEnumerationTask` were implemented closely based off their qualitative counterparts `SimpleTask1` and `SimpleTask2`, respectively, preserving the same action flow and style of reporting, but with data associated to the quantitative side instead.

In order to support the metrics of the new types of solvers and so on, an extension `A4Reporter` ← `A4QtReporter` was established, which alongside the kinds of messages supported already in a qualitative scenario, it is also equipped with the following:

- `translate` indicates that the SMT Problem/PRISM model associated with the Kodkod problem at hand is about to be generated, as well as reports a few of the solving details, like the context of the analysis being performed and which quantitative solver will be used;
- `smt2` is used to report the `SMTStatistics` associated with the SMT specification derived, as well as the time it took to determine the latter;
- `prism` has the same purpose as `smt2` but for problems where the PRISM solver is used instead, presenting `PrismStatistics` about the translation process;
- `resultUNKNOWN` is added to fully support the possible outcomes of an SMT Solver, used to report the UNKNOWN judgment similarly to the existing `resultSAT` and `resultUNSAT` methods.

This kind of reporter is used within the `solve` method implemented in `A4Solution` for quantitative problems described previously. Moreover, the concrete implementation of the reporter is done in `QuantitativeTask`, taking advantage of the `callbacks` used to send messages to the GUI so that they can be perceived by the user.

Now, once the `QuantitativeTask` is executed by the `SimpleGUI`, the model at hand is solved as follows:

1. Parse the `.als` specification written by the modeler.
2. The Alloy model and the target command(s) are translated into the adequate Kodkod problem through `TranslateQTAlloyToKodkod`, which alongside the `A4Options` provided by `SimpleGUI` and an instance of `A4QtReporter` are passed to the `solve` method from `A4Solution`, determining the satisfiability of the problem with respect to the solving options as well as an instance in case SAT was the outcome. As already mentioned, during this process the reporter is used to provide translation metrics.
3. The satisfiable command(s) have their `A4Solution` object preserved in XML format by the `A4SolutionWriter`, to be later handled by the analysis features offered by the Alloy Analyzer.
4. Finally, the task ends by presenting the solving outcome and metrics for each command considered.

When attempting to find more instances of the problem at hand, `QTEnumerationTask` will request the next solution to the `A4Solution` associated with the current model. After solving for further solutions, if a SAT solution is obtained, then it is written into a XML file and the instance found can be studied by the modeler with the help of the analyzer; in case an SMT solver is being used and the UNKNOWN response is given, then the user is notified that there *may or may not* be further instances for this model; lastly, an UNSAT response indicates that there are no more solutions to the problem.

VISUALIZER After successfully finding an instance of the Alloy model, it can be presented graphically to the user via the `VizGUI`, which was also subject to a few modifications in order to be capable of properly picturing a quantitative Alloy instance.

`SimpleGUI` is responsible for notifying the `VizGUI` of the analysis context being considered at a given moment in time, in particular, when the user requests the visualizer over a quantitative model, the `A4Solution` will be loaded from the XML format created by a quantitative task through the quantitative extension of `A4SolutionReader`, while in a qualitative scenario, the XML does not contain measurable data associated with the instance and thus the regular `A4SolutionReader` is used instead.

Furthermore, all the helper classes and structures used by the `VizGUI` to manage an Alloy instance suffered minor adjustments to be able to take the quantitative information into account, in a obvious fashion.

From all the different visual representation types supported by the `VizGUI`, the following were successfully extended to support quantitative instances:

- Text (Figure 6.2)

```
---INSTANCE---
integers={}
univ={A$0, A$1, B$0, B$1, C$0}
Int={}
seq/Int={0, 1}
String={}
none={}
this/A={A$0, A$1}
this/A<x={ (A$0->B$0,2), (A$0->B$1,4), (A$1->B$1,4) }
this/A<y={ (A$1->C$0,4) }
this/A<z={A$0->B$0->A$1}
this/B={B$0, B$1}
this/C={C$0}
```

Figure 6.2: Txt representation of an Alloy instance.

- Table (Figure 6.3)

this/B
B ⁰
B ¹

this/A	x	y	z
A ⁰	B ⁰	2	B ⁰ A ¹
	B ¹	4	
A ¹	B ¹	4	C ⁰ 4

Figure 6.3: Table representation of an Alloy instance.

- Graph (Figure 6.4)

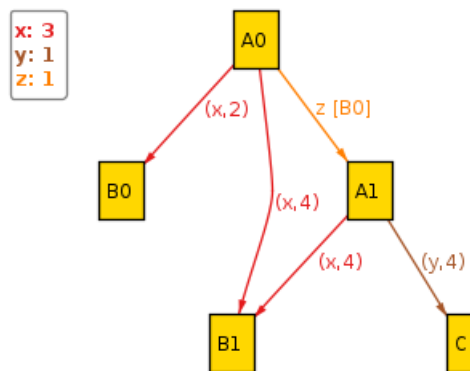


Figure 6.4: Viz representation of an Alloy instance.

The edges now display the weight associated to the tuple through a pair (R, w) for tuples within a quantitative relation.

When working over the probabilistic setting, there may be w of various lengths, potentially cluttering the visualizer and thus, to improve readability every probability will be pre-processed before being added to the graph, trimming the value at 3 decimal places maximum or rounding the value to 0 if $w < 0.001$. However the user is always able to view the full exact values through the Text representation or in the Evaluator.

For every kind of representation, if the relation is represented by a Boolean matrix, then it is depicted exactly the same as in qualitative instances, however, relations described by a numeric matrix have the weight of each tuple explicitly represented accordingly, as exemplified before.

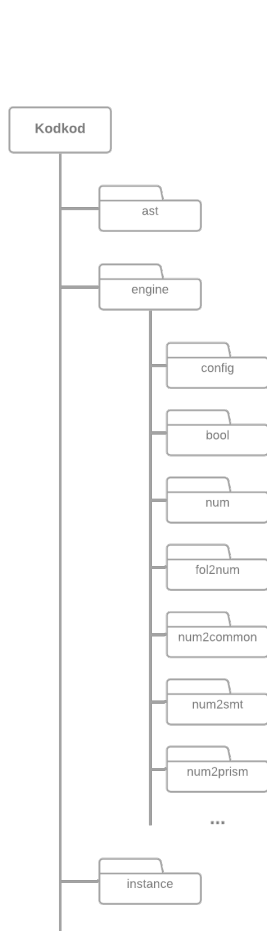
EVALUATOR Providing an evaluator over quantitative instances at this stage proceeded rather smoothly, given all the groundwork made prior: a quantitative evaluator was implemented in the Kodkod extension as mentioned previously, which in turn can be accessed by the methods considered in `A4Solution`.

Thus, `SimpleGUI` loads the `A4Solution` from its XML representation adequately using the quantitative extension of `A4SolutionReader` when handling quantitative instances, parses the Alloy expression, which is then passed to the `A4Solution` evaluation method, calculating the result using the quantitative evaluator, to be properly represented through Alloy structures, which in turn is presented to the user in the same way as its qualitative counterpart.

With that, a quantitative extension to Alloy and the Alloy Analyzer was successfully achieved.

6.3 PROJECT STRUCTURE

During the development of the quantitative extension for both Kodkod and Alloy, the original Alloy project suffered changes, either by adding new components or modifying existing ones to accommodate to the requirements defined during this dissertation. In the end, the resulting structure associated with the extension is organized as follows:



KODKOD

AST The Kodkod abstract syntax was extended to support new operators and the specification of quantitative constraints.

ENGINE The workflow during quantitative solving was defined at this point, adding the capability of requesting quantitative solving over Kodkod problems, producing quantitative instances, and further analyse the solutions obtained.

CONFIG Introduce customizable solving options associated with the quantitative solvers supported.

NUM Creation and management of the life cycle of the numeric structures used to represent Numeric Circuits – numeric values, matrices, gates, ...

FOL2NUM Handles the translation between a Kodkod problem and its NC representation, for either integer or probabilistic settings.

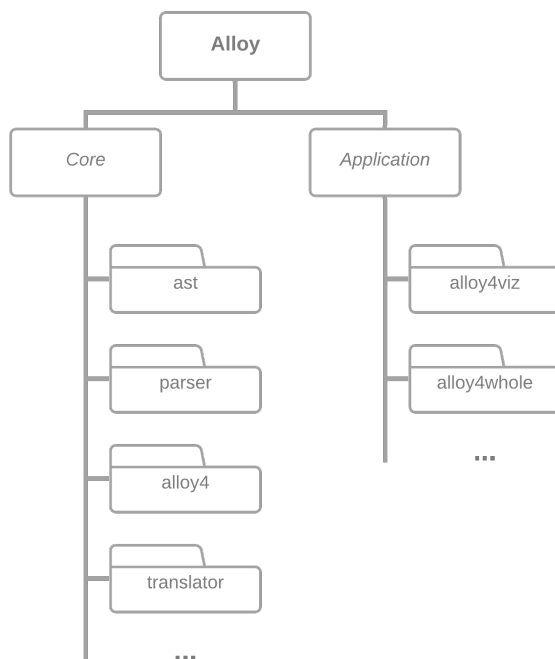
NUM2COMMON Helper methods and structures to handle quantitative solvers and the translations produced at a more general level.

NUM2SMT Manages everything related to the SMT Solvers, including the correspondence from a NC and the respective SMT problem, interaction with the SMT Solvers supported and the outcomes obtained.

NUM2PRISM Similarly to *num2smt* but focused on PRISM instead, handles the PRISM model associated with the Kodkod problem at hand, executes model checking of the last and processes the satisfiability results.

INSTANCE Added support to create and represent quantitative Kodkod instances.

ALLOY



- Core

AST To support the Kodkod syntax extension, the Alloy AST itself was modified to also support new operators and quantitative relational expressions.

PARSER Adapted the parser to support the language extension.

ALLOY4 Miscellaneous components of Alloy were adjusted at this point, like the introduction of a reporter for quantitative solving, new solving preferences, and so on.

TRANSLATOR Is the bridge between Alloy and Kodkod, where the Kodkod problem is properly generated, the solving process is handled and an Alloy solution is produced.

- Application

ALLOY4VIZ Adaptation of the visualizer to be able to display quantitative instances.

ALLOY4WHOLE Where the main application workflow is controlled to accommodate to both kinds of analysis, launching the newly implemented quantitative task as well as the visualizer and evaluator accordingly.

6.4 WORKFLOW

In contrast to the architecture proposed initially in Section 4.3.1, the actual implementation ended up functioning as pictured in Figure 6.5.

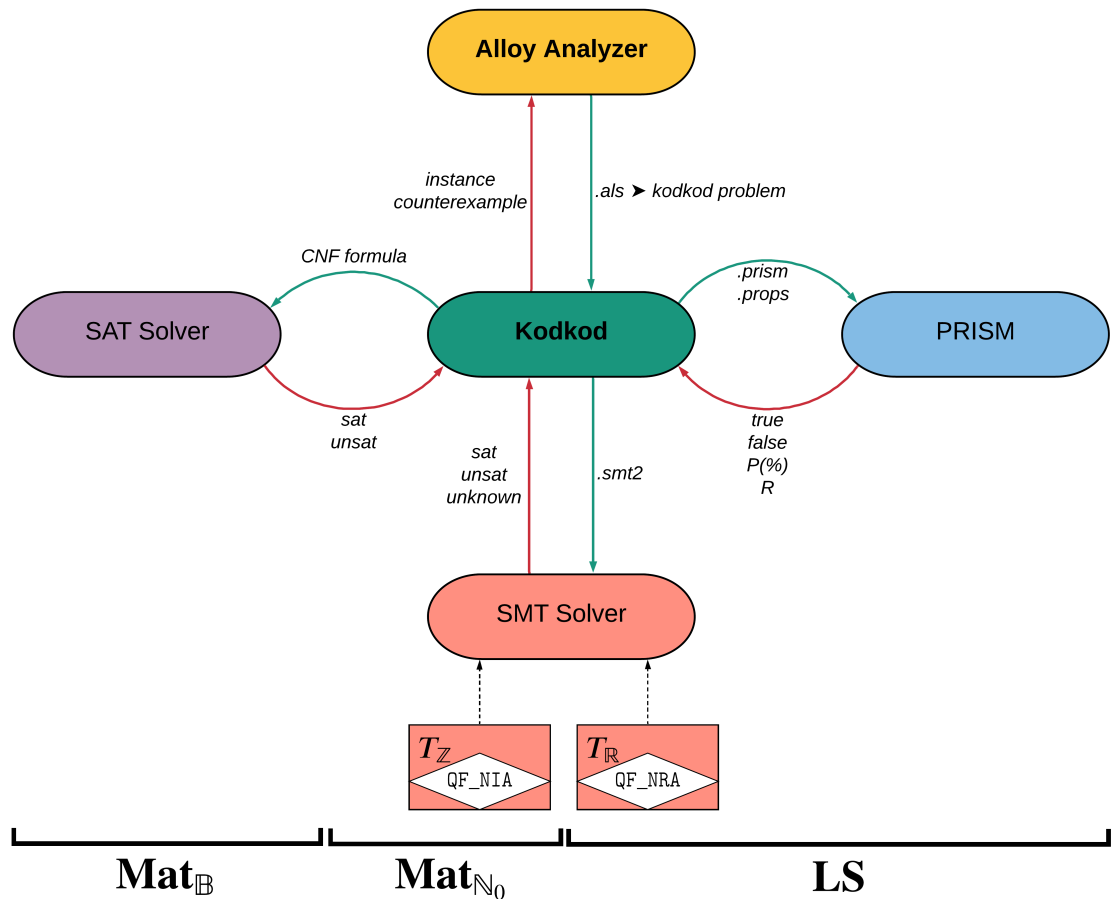


Figure 6.5: Quantitative Alloy Architecture.

As intended, the original Alloy capabilities are still in use, through the usual SAT Solvers. Furthermore, PRISM is able to handle probabilistic systems as predicted. However, the capabilities of SMT Solvers surpassed the initial expectations, making them responsible for not only numeric problems, but also being available to the user alongside PRISM when modeling systems which display faulty behaviour, having chosen the theory of integers (fragment QF_NIA) as the background theory to handle $Mat_{\mathbb{N}_0}$ and the theory of reals (logic QF_NRA) when dealing with LS .

Finally, the process happening under Alloy when performing quantitative solving, through the extension detailed during this chapter, is further illustrated in Figure 6.6.

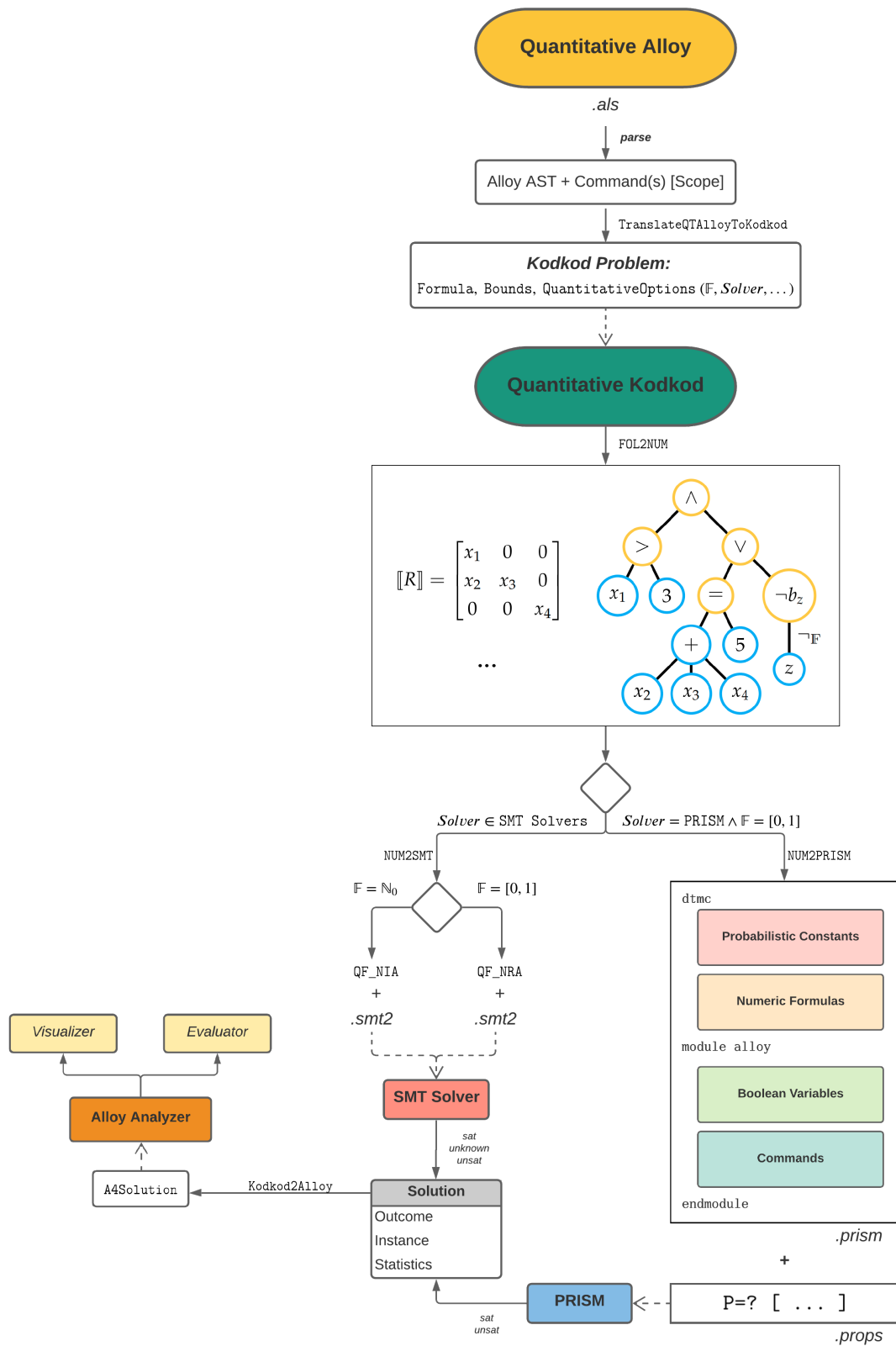


Figure 6.6: Quantitative Alloy Workflow.

6.5 QUANTITATIVE ALLOY IN PRACTICE

It is now possible to perform quantitative analysis using Alloy, thus, this section will focus on expanding on how to take advantage of the new features offered, present some of the different kinds of quantitative constraints supported, the way of managing domain specific information (when \mathbb{F} represents either the natural numbers or the probabilistic setting), how to execute quantitative analysis, as well as study some of the extension capabilities.

ALLOY GUI To freely switch between the quantitative tools to be used during solving, as well as the domain of the latter, the supported alternatives were added to the GUI of the application. Figure 6.7 for instance, shows that the model will be analysed for $\mathbb{F} = \mathbb{N}_0$ using CVC4 Solver, as preferred under “SMT Solver”; while if the “Analysis Context” is swapped to “Probabilistic”, then the “Probabilistic Solver” will be used instead, which corresponds to PRISM in this example. Furthermore, once the “Boolean” context is selected, then the solving options correspond to those specified under the SAT solving configurations, the same way as in the original Alloy.

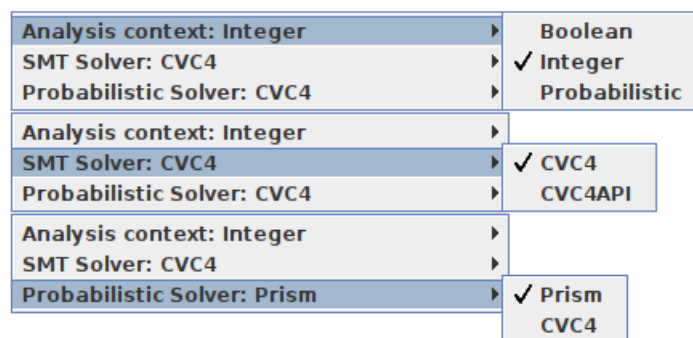


Figure 6.7: Quantitative Solving Options.

Having specified a *.als* specification as well as a command, by pressing *Execute* the solving process begins. Figure 6.8 shows the way that the solving details and outcome are presented to the user when taking advantage of an SMT solver at the top, and PRISM at the bottom.

Given a satisfiable model, it can be further studied using the Alloy Visualizer and/or Evaluator, as shown previously in Figures 5.3, 6.2, 6.3 and 6.4 for example.

QUANTITATIVE CONSTRAINTS By taking advantage of the additions to the Alloy language, new kinds of constraints can now be specified.

Tuple weights are handled through the cardinality operator $\#$, which measures the sum of every weight of its operand. Thus, if the weight associated with a tuple is already known beforehand, it can be imposed within the Alloy specification similarly to the following example:

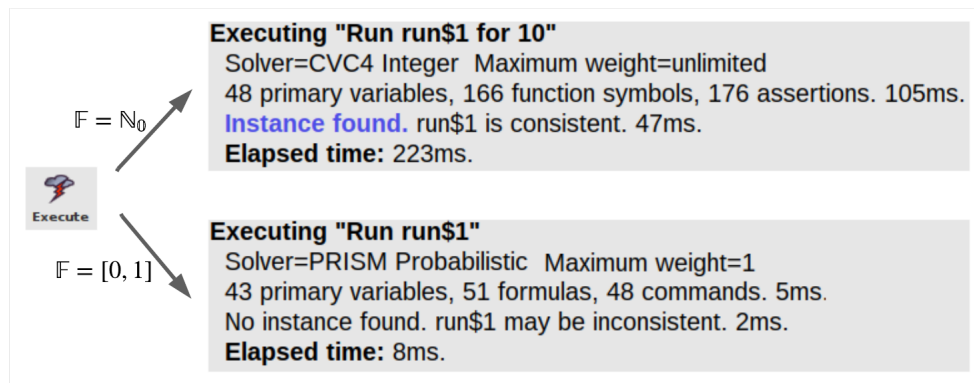


Figure 6.8: Example of the execution of a satisfiable command under the Integer context solved using an SMT Solver and an unsatisfiable result for a probabilistic model solved with PRISM.

```
sig B{}
sig A{ R : set B }
lone sig X extends A{} // X is a known potential "atom" of A
lone sig Y extends B{} // Y is a known potential "atom" of B

// Tuple (X, Y) must occur in R with weight 3
fact{ #(X.R :=> Y) = 3 }
```

Other constraints over the weight of a certain tuple can be encoded analogously, accessing the weight of the target focus in such way.

Naturally, these kinds of constraints over relation weights can be more broad, by taking X and/or Y as sets for instance. Table 6.3 describes step-by-step the evaluation of such kind of formula over the tuple's weights. Alternatively, if both X and Y are relations of arity 1, then the weight of the tuple formed by them on a certain relation can also be accessed by expressions like $\#(X <: R :=> Y)$. Do note that for expressions of the previous form, $\#(X <: R.Y)$ and $\#(X.R :=> Y)$, require X and Y to be unary relations, respectively, given the definition of the domain/range operators.

S	{ (a->x, 2), (a->z, 3), (b->y, 4), (b->w, 1), (b->z, 1) }
S.Q	{ (a, 3), (b, 5) }
b <: S.Q	{ (b, 5) }
$\#(b <: S.Q)$	5
$\#(b <: S.Q) = 10$	false

Table 6.3: Evaluating $\#(b <: S.Q) = 10$ with $Q = \{ y, z \}$.

The addition of drop to the language allows the modeler to deal with relations over $Mat_{\mathbb{B}}$, which can be used to reason over reachability properties like during qualitative analysis, but under a quantitative setting

instead. For example, under $\mathbb{F} = \mathbb{N}_0$, $\#R < 4$ requires the sum of the weights associated with the tuples of R to add up to 3 and thus, $R = \{ a_0 \rightarrow b_1, a_1 \rightarrow b_0 \}$, $R = \{ (a_0 \rightarrow b_0, 3) \}$ and so on, are possible interpretations of R that abide to that constraint. Now, if $\#(\text{drop } R) < 4$ is imposed instead, then R may contain, at maximum, 3 different tuples. Since `drop` effectively “discards” weights, they are not bound like in the previous version, meaning that $R = \{ (a_0 \rightarrow b_1, 15), (a_1 \rightarrow b_1, 2) \}$ would obey this constraint, even though it would violate the one discussed above, as $\#R = 17$. Therefore, by combining the line of thought considered before to handle tuples, together with the use of `drop` in this manner, one can encode reachability properties precisely. Taking R as a more complex relational expression in this fashion, for instance, representing a particular selection, category, group and the like, then $\#(\text{drop } R) < 4$ would accept a maximum of 3 tuples belonging to that group, similarly to the constraint considered within the model to be presented in Section 7.4.

Despite the capability of handling weighted relations being the main focus of this extension, in some situations it may be useful or even necessary to use ordinary relations in the model at hand. To represent such kind of relation in a quantitative setting `drop` can be used, by specifying the requirement $R = \text{drop } R$, that is, demand that the relation R coincides with its representation from the Boolean perspective.

Unary relations with meaningful weights can be successfully specified exactly as shown previously in Figures 5.2, 5.3. Like mentioned in Section 5.2, `drop` can also be composed with multiplicity keywords to describe interesting properties.

INTEGER SCOPE Putting the integer scope into practice, as shown in Section 5.4, it can be used to reduce significantly the number of potential instances for a given model. Since the tool is now working over an infinite domain, some Alloy specifications may present an infinite number of satisfiable solutions and thus, by imposing an upper bound to the acceptable weights, that number can be reduced to a more manageable finite amount. Figure 6.9 shows two commands, one with and the other without using integer scope. Without

<pre>run noIntScope{} run withIntScope{} for 6 Int</pre>	<pre>Executing "Run noIntScope" Solver=CVC4 Integer Maximum weight=unlimited 2 primary variables, 8 function symbols, 8 assertions. 4ms. Instance found. noIntScope is consistent. 7ms. Elapsed time: 13ms.</pre>	<pre>Executing "Run withIntScope for 6 int" Solver=CVC4 Integer Maximum weight=6 2 primary variables, 8 function symbols, 9 assertions. 4ms. Instance found. withIntScope is consistent. 5ms. Elapsed time: 11ms.</pre>
--	---	---

Figure 6.9: Example of a command with and without imposing an integer scope.

the help of an integer scope, during the enumeration of solutions, if at least one tuple within a given relation can freely vary its weight without compromising the satisfiability of solutions, the tool may end up producing solutions containing the exact same tuples, only changing the weight of that tuple indefinitely, which may or may not be interesting. Thus, by imposing such limit, the Alloy Analyzer will be forced to find solutions with further differences from the previous.

PROBABILITY Neither the original Alloy language nor the current version of the extension supports the specification of non-integer numbers explicitly.

However, in order to model probabilistic problems, it is necessary to be able to represent other kinds of numbers. Turns out that it was immediately possible to do so after implementing the quantitative extension, through the help of existing operators, once again, due to the “keep definition, change category” (Oliveira and Miraldo, 2016) approach taken. As mentioned previously, the integers supported in the original Alloy, within a quantitative setting, represent natural numbers if $\mathbb{F} = \mathbb{N}_0$ and real numbers for $\mathbb{F} = [0, 1]$, having operations between integers also shifted to the other domains. Therefore, by using `fun/mul` and `fun/div2` which, besides representing the Hadamard product and division in the extension, correspond to integer multiplication and division, one is able to describe real numbered values, in particular, probabilities, for example, 31.7% can be specified as `317 fun/div 1000` within an `.als` model, and also on the evaluator like Figure 6.10 shows.

```

317 fun/div 1000
0.317
2 fun/mul (8 fun/div 100)
0.16
15 fun/div 7
2.142857142857143

```

Figure 6.10: Example of using the Alloy Evaluator to handle real valued numbers.

PROBABILISTIC CONTRACTS With the newly acquired expressiveness from extending Alloy to the quantitative realm, it is now possible to measure probabilistic contracts using the Alloy Analyzer.

Recall the definition of a probabilistic contract: Given any probabilistic function $f : A \rightarrow \mathcal{DB}$,

```
sig A{ f : one B }
```

```
sig B{ }
```

the chance of the probabilistic contract $q \xleftarrow{f} p$ associated with the precondition p and postcondition q holding, depends on the input distribution $\delta : 1 \rightarrow A$,

```
one sig Unit{ delta : one A }
```

Knowing δ , the probabilistic contract can be measured as follows (2.41):

$$\llbracket q \xleftarrow{f} p \rrbracket_{\delta} = (q \cdot f \times p) \cdot \frac{\delta}{p \cdot \delta}$$

which can now be encoded in Alloy as:

² Other utility functions like `mul` and `div` can also be used, as they are defined using `fun/mul` and `fun/div`, but unlike them, which return the integer value, they return a subset of the signature `Int`, e.g., `2 fun/mul 3 = 6` and `mul[2,3] = { 6 }`, which may be useful depending on the situation.


```

fun measureContract[p' : A, q' : B] : Int {
  let p = drop p', q = drop q' | #delta.(f.q fun/mul p) fun/div #delta.p
}

```

Note that both p and q given are subject to *drop*, to ensure that they represent valid Boolean vectors.

Let us consider the probabilistic function studied by Oliveira (2017a) as an example:

f	a_1	a_2	a_3
b_1	0.7	0.01	1
b_2	0.3	0.99	0

which in turn is specified as

```

fact PF{
  #(a1.f :=> b1) = div[7, 10]
  #(a2.f :=> b1) = div[1, 100]
  one a3.f :=> b1
}

```

By fixing an input distribution,

	δ
a_1	0.1
a_2	0.2
a_3	0.7

in Alloy,

```

run knownDelta{
  #(delta :=> a1) = div[1, 10]
  #(delta :=> a2) = div[2, 10]
  // #(delta :=> a3) = div[7, 10] // Does not need to be explicitly
  // required since it is a distribution
  // (its probabilities must add up to 1)
}

```

then there is a single solution to the model pictured in Figure 6.11.

Moreover, one is fully able to measure the probability of the contract holding with respect to that δ through the quantitative extension of Alloy, as can be seen in Figure 6.12. These contracts were the same as those

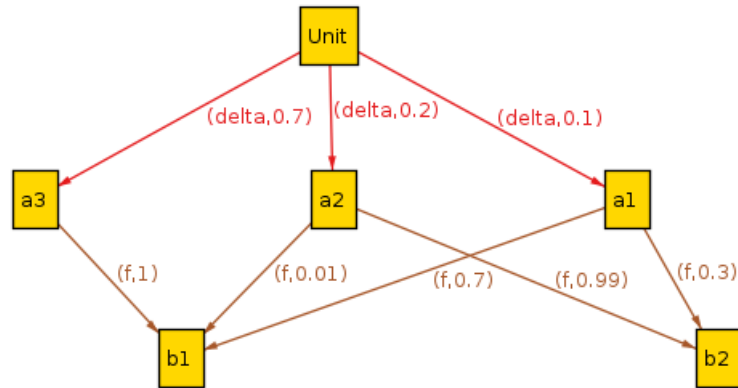


Figure 6.11: Instance determined for the constant f and δ considered.

```

#measureContract[a1 + a2, b2]
0.7599999999999999
#measureContract[a3, b2]
0
#measureContract[A, b2]
0.228
#measureContract[a1 + a2, B]
1

```

Figure 6.12: Using the Alloy Evaluator to measure the probability of different contracts with respect to the instance's f and δ .

studied by Oliveira (2017a), whose values evaluate to:

$$\{b_2\} \xleftarrow{f} \{a_1, a_2\} = 76\%$$

$$\{b_2\} \xleftarrow{f} \{a_3\} = 0\%$$

$$\{b_2\} \xleftarrow{f} true = 22.8\%$$

$$true \xleftarrow{f} \{a_1, a_2\} = 100\%$$

precisely the same output given by the Analyzer, as intended.

Now, the power of this tool shines once δ is unknown. It would be useful to study the validity of a contract in general, and take advantage of Alloy's model finding nature to determine if, for instance, the contract $\{b_2\} \xleftarrow{f} \{a_1, a_2\}$ holds with, at least, 80% probability, no matter the input distribution.

Turns out that the extension is capable of doing exactly that, similarly to the proposed check command:

```
assert contract{
```

```

    measureContract[a1 + a2, b2] >= 8 fun/div 10
  }
check contract

```

After checking, Alloy was able to find examples of δ for which the contract holds with less than 80% chance, for example, given the δ presented in Figure 6.13, the contract holds with 71.6% chance.

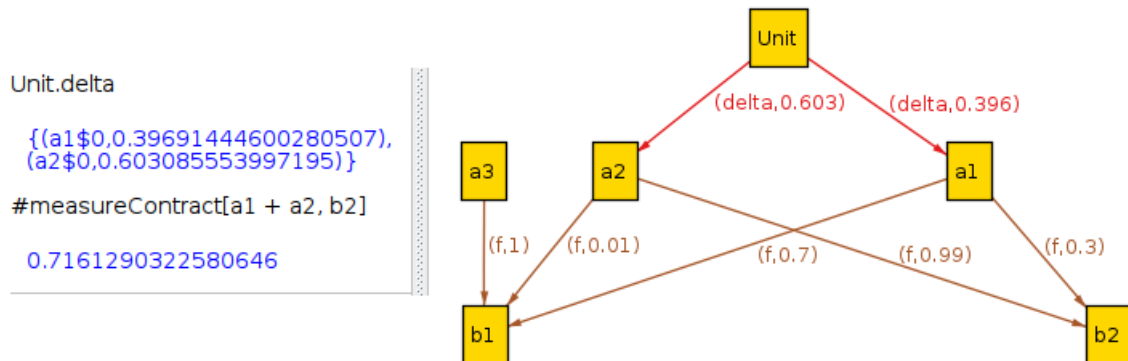


Figure 6.13: A δ determined by Alloy as a counterexample when checking the contract delimited by $p = \{a_1, a_2\}$ and $q = \{b_2\}$ over f for a probability lower than 80%.

Furthermore, it is possible to estimate the minimum probability for which the contract can hold given any δ , adjusting the probability in the check being performed, until Alloy is no longer able to find a single δ for which the probability is lower. For this particular example, that probability is around 30%, like pictured in Figure 6.14.

```

assert contract301{
  measureContract[a1 + a2, b2] >= div[301, 1000]
}
check contract301

assert contract30{
  measureContract[a1 + a2, b2] >= div[3, 10]
}
check contract30

```

Executing "Check contract301"
 Solver=CVC4 Probabilistic Maximum weight=1
 15 primary variables, 118 function symbols, 123 assertions. 12ms.

Executing "Check contract30"
 Solver=CVC4 Probabilistic Maximum weight=1
 15 primary variables, 118 function symbols, 123 assertions. 14ms.

2 commands were executed. The results are:
 #1: Counterexample found. contract301 is invalid. 21ms.
 #2: No counterexample found. contract30 may be valid. 35ms.

Elapsed time: 87ms.

Figure 6.14: The contract specified with $p = \{a_1, a_2\}$ and $q = \{b_2\}$ holds for the given f with, at least, 30% chance for every possible δ .

Further capabilities of this extension will be closely studied when attempting to tackle the case studies considered for this project in the next chapter.

6.6 SUMMARY

Enhancing Alloy and Kodkod to be able to handle measurable data was the main focus on this chapter.

It begins by addressing the development of a quantitative Kodkod extension: given a Numeric Circuit built from a quantitative Kodkod problem as established in the previous chapter, the next step is to generate an adequate specification to be fed into a quantitative solver and thus, the integration of CVC4 and PRISM was detailed, including the need to setup a correspondence between the numeric structures and an SMT problem (*.smt2*) or a PRISM model (*.prism* together with *.props*), also studied thoroughly. Finally, the quantitative extension of Kodkod is accomplished by establishing the production of a Kodkod instance from a quantitative outcome.

This chapter then goes over the adjustments made to Alloy and its Analyzer, so that its quantitative extension could be achieved by conforming to quantitative Kodkod. First, the Alloy language had to be extended to support further quantitative constraints; then, the differences of the correspondence between the Alloy specification at hand and the respective Kodkod problem under a qualitative or a quantitative context were detailed; a quantitative Alloy solution is then derived from a quantitative Kodkod instance. The implementation of quantitative Alloy is then concluded after the adaptation of the features provided by the Alloy Analyzer – the Visualizer and the Evaluator.

With the extension successfully developed, the final system architecture and its workflow are highlighted.

In the end, the chapter details how to take advantage of the quantitative features, at both specification level and application level.

CASE STUDIES

With the quantitative extension of Alloy complete, the next step is to assess its power by handling the case studies previously introduced in Section 4.1.

Each case study will be subject to quantitative analysis on different levels:

- **Implicit Quantification** over the bibliometrics case study, that is, extracting measurable data from existing alloy models with minimal to no modifications to the specification, simply by taking advantage of the lack of idempotency of morphism addition.
- Perform **Explicit Quantification** – that is, through the use of **quantitative invariants** or by imposing constraints that work at weight-level explicitly on the specification or by taking advantage of the quantitative extension of the Alloy language – in the football championship setting to model the quantitative relation *History*;
- Model the Bayesian Network using Alloy syntax and execute **probabilistic analysis** over it, as well as introduce **probabilistic contract** concepts explicitly in Alloy.

Then, this chapter makes a throwback to the existing work which addresses the same area as this dissertation, namely by attempting to use the Alloy extension to model those examples.

Finally, it presents a benchmark to evaluate the usability of the tool, as well as to check how it behaves comparatively to some existing alternatives.

7.1 BIBLIOMETRICS

Initially, Section 4.1.1 presents a relational model of a simple bibliographic database, from which one is unable to handle measurable data. Later on, by studying it under *Mat* lenses, bibliographic metrics were immediately drawn out, provoking the thought: will the quantitative extension of Alloy be instantly able to gather quantitative data from already existing models?

Figure 7.1 displays an instance when performing quantitative analysis of the specification present in Appendix A.1.1. Alloy was, indeed, able to bring out quantitative information from the original specification

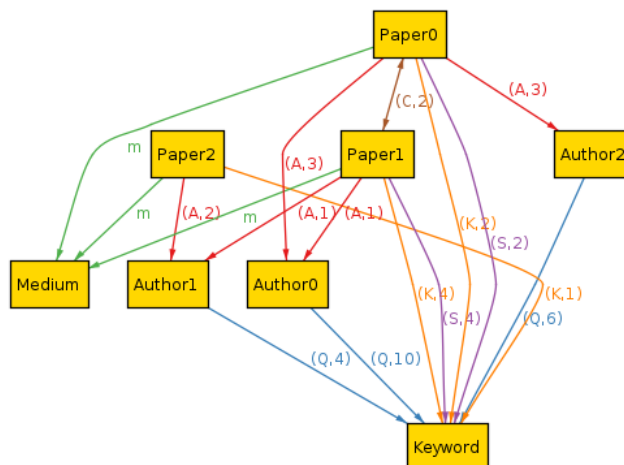


Figure 7.1: A quantitative instance of the original bibliographic system Alloy model.

– *too much* actually. Naturally, not only the relations S and Q which were analysed under quantitative semantics in Chapter 4, but every relation which is not properly constrained, will now be represented by a numeric matrix, namely, relations C , K and A , coincidentally, all of which should still be represented by a Boolean matrix as:

- A relation A which describes who the authors of a given paper are does not require quantitative weights, an author either participated in the writing of a paper or did not, sufficiently modeled under \mathbb{B} .
- Analogously, the keywords K of a given paper do not bring any useful information associated with weights outside of the $\{0, 1\}$ range.
- Arguably, C could potentially make sense as a quantitative relation, counting the number of times that a citation of a paper is referred to within the text of the paper at hand. Nonetheless, when measuring bibliometrics typically a citation is always counted as 1, no matter how many times it is cited on the paper, meaning that it is also properly modeled as an ordinary relation.

While during qualitative analysis, not being able to extract measurable data was the struggle, now the problem is mirrored, as there are too many elements of the model being quantified. However, in contrast, the quantitative setting has the upper hand of higher expressiveness when compared to the Boolean context, having access to the tools needed to also reason over qualitative relations.

Having set the boundaries of which relations are to be represented by a Boolean matrix or a numeric matrix, the original specification can be extended to impose constraints that “bring down” the relevant relations to $Mat_{\mathbb{B}}$, like seen previously in Section 6.5:

```
fact ordinaryRelations{
```

```

K = drop K
A = drop A
C = drop C
}

```

Now that K is a relation described by Boolean values, a new problem arises: S will, by consequence, be represented by a Boolean matrix, which is unavoidable when $S = K \cap K \cdot C^\circ$. Taking into account the implementation of \cap in the quantitative extension of Alloy, described in Section 5.5, each tuple of S is a tuple which occurs on both K and $K \cdot C^\circ$, assuming the **minimum**-valued weight between the two. Since the weight of the tuples within K will never be outside the $\{0, 1\}$ value range, then the weights of S will also never exceed 1, meaning that the quantitative information arising from $K \cdot C^\circ$ will be lost.

To achieve the quantitative interpretation of S desired, its weights have to abide to those associated with $K \cdot C^\circ$ while, at the same time, making sure that $k S p$ only counts the papers which cite paper p in the same area k , that is, reachability wise coincides with $K \cap K \cdot C^\circ$. Therefore, S will now be specified by the following requirements:

- The Boolean representation of S must coincide with the initial definition, i.e., relating a given paper with the keywords that identify the area on which it was cited.

$$\text{drop } S = K \cap K \cdot C^\circ$$

- Counts how many papers cited the paper in focus, within the same area.

$$\forall i. \llbracket S \rrbracket[i] \neq 0 \Rightarrow \llbracket S \rrbracket[i] = \llbracket K \cdot C^\circ \rrbracket[i]$$

specified through the Alloy extension as:

```

drop S = K & ~C.K
all p : Paper, k : Keyword |
  (p -> k) in S implies #(p <: S.k) = #(p <: (~C.K).k)

```

At this point, the Alloy specification properly models the bibliographic system under a quantitative context equivalently to the original model within the qualitative setting. Figure 7.2 highlights exactly that: when an instance found by a SAT solver is subject to a quantitative tool, measurable data is immediately derived,

- Every article belongs to the same area (Keyword);
- C – Paper1 cites Paper0 and Paper2 cites Paper0 and Paper1;
- S – Thus Paper0 is cited twice and Paper1 is cited once within the same area;

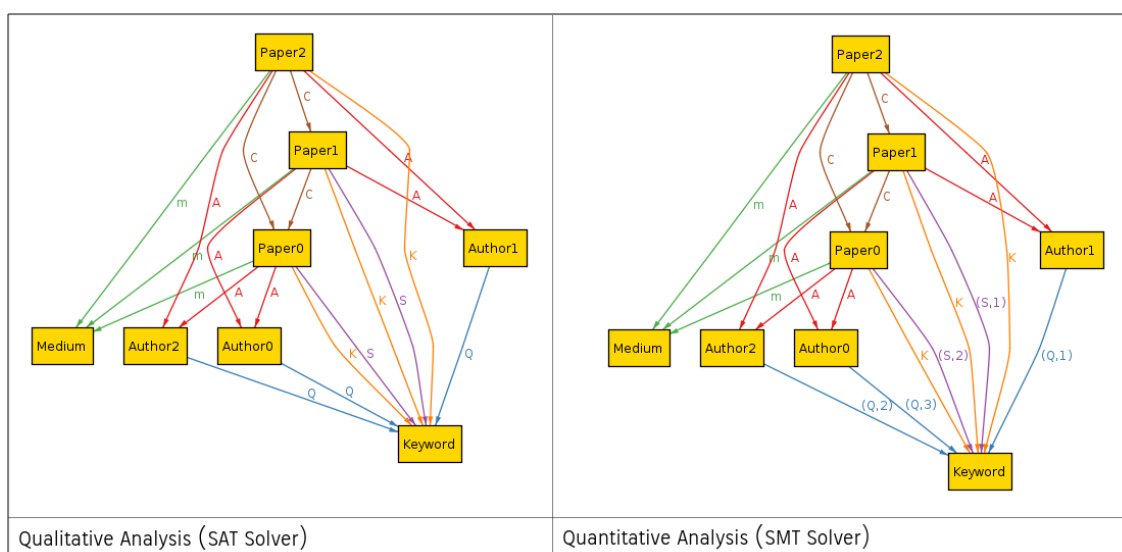


Figure 7.2: The same Alloy solution (structure wise) when interpreted under Boolean versus quantitative semantics.

- A – Paper0 was written by Author0 and Author2 while Author0 and Author1 are the authors of Paper1.
- Q – Therefore, since Author1 is one of the authors of Paper1 which was cited once, this author received one citation; similarly, Author2 is the author of Paper2 being referred twice, meaning that so did the author; due to Author0 being one of the authors of both Paper0 and Paper1 then, in total, this author received three citations.

This information was already present in the original qualitative model of the bibliographic database, but the data was nowhere to be seen. When subject to quantitative analysis, the same information arose spontaneously, being able to be reasoned over explicitly. Thus, **implicit quantification** was achieved using the Alloy extension.

By increasing the scope, further interesting instances can be found where the analytics extracted in that scale are more evident. Figure 7.3 is an example of another solution found through quantitative solving (the full solution in text representation is present in Appendix A.1.3). The measurable data can then be further reasoned with, organized and so on as the user desires, for instance, by taking advantage of other tools: Figure 7.4 shows a graphic with the bibliometrics extracted from the Q relation calculated for this instance.

Lastly, all there is left to do is to try to describe the quantitative relation $Z = \frac{Q}{S.T}$, which through division measures the impact of each author within every area in the big picture. With the extended language, Alloy possesses enough expressiveness to encode such a relation, as follows:

$$Z = Q \text{ fun/div (Author} \rightarrow \text{Paper)}.S$$

Unfortunately, since the analysis is being performed under $Mat_{\mathbb{N}_0}$, Z is represented by a matrix of natural numbers, with the division in question being the Euclidean division, and given the scale of the percentiles

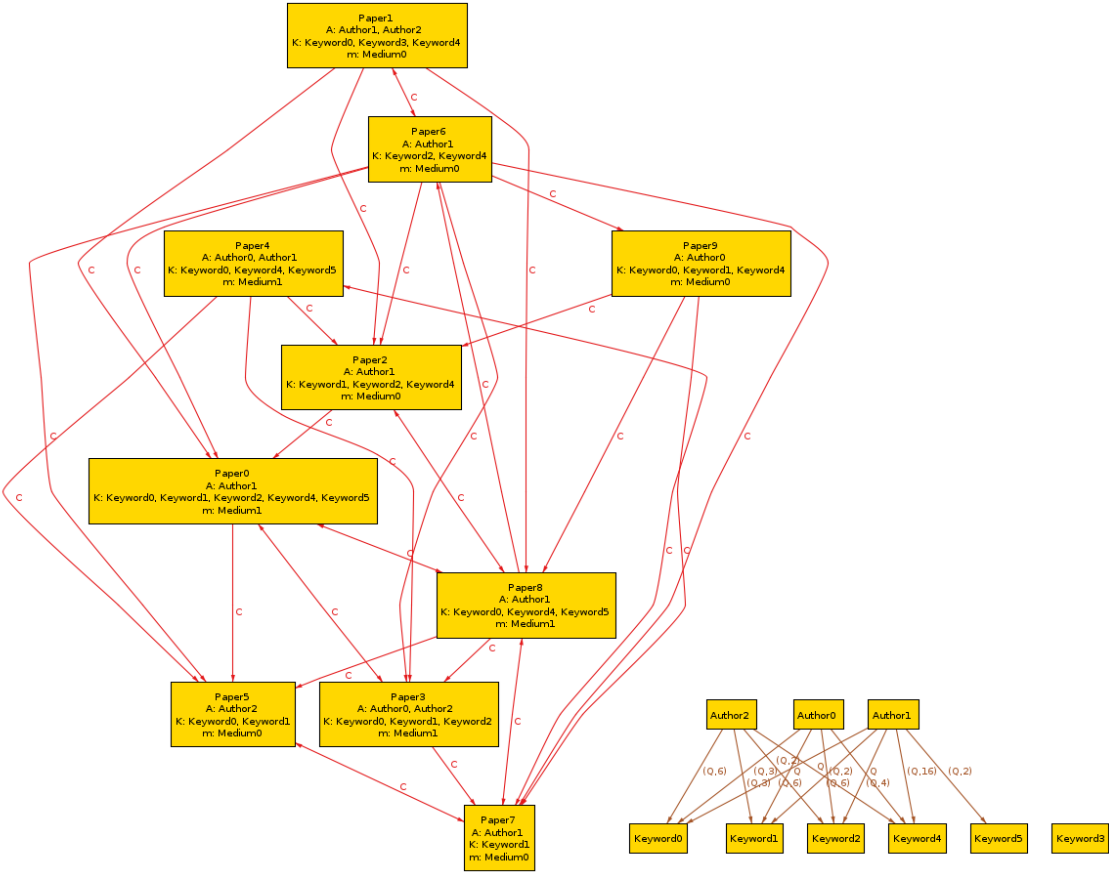


Figure 7.3: Example of a quantitative instance for the bibliographic system.

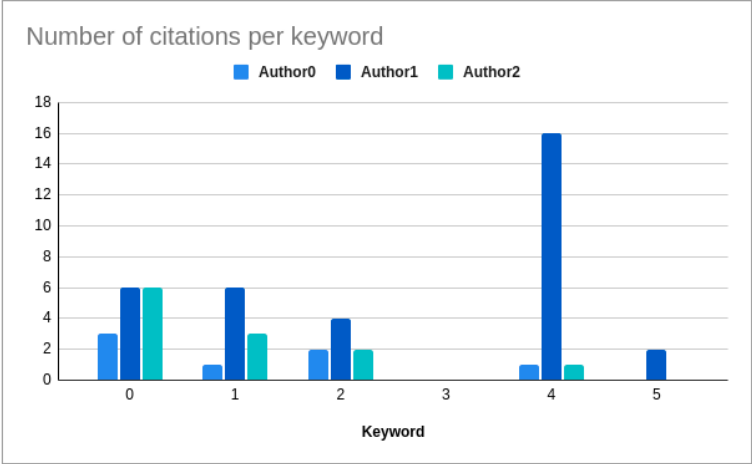


Figure 7.4: Number of citations each author received per area for the instance presented in Figure 7.3.

being calculated, i.e., each weight of Z will belong to the $[0, 1]$ range, then under this domain only instances where Z is empty or contains tuple(s) with weight 1 (meaning that every author within Z took part in writing

all the papers that were cited within the same area) will be provided by Alloy, which are not very interesting nor do they properly represent Z as intended.

However, this does pique the interest: what would it take to overcome such limitation and make Alloy able to handle Z precisely. It immediately comes to mind considering a broader domain where the kind of division desired could be applied and, at the same time, still be able to support this model as a whole. Perhaps a new kind of analysis could be added to Alloy, involving another class of matrices also, matrices of positive real numbers $Mat_{\mathbb{R}_0^+}$ for instance, where division would now be real numbered division, which is able to evaluate data over the $[0, 1]$ interval explicitly. Figure 7.5 displays an instance produced within

this/Author	Q		Z	
Author ⁰	Keyword ⁰	3.0	Keyword ⁰	0.75
Author ¹	Keyword ⁰	1	Keyword ⁰	0.25
Author ²	Keyword ⁰	2.0	Keyword ⁰	0.5

this/Paper	C	K	A	m	S	
Paper ⁰		Keyword ⁰	Author ⁰	Medium ⁰	Keyword ⁰	2.0
Paper ¹	Paper ⁰	Keyword ⁰	Author ⁰	Medium ⁰	Keyword ⁰	1
	Paper ²		Author ²			
Paper ²	Paper ⁰	Keyword ⁰	Author ¹	Medium ⁰	Keyword ⁰	1
	Paper ¹		Author ²			

Alloy Evaluator

```

S
{
  (Paper$0->Keyword$0.2.0),
  (Paper$1->Keyword$0.1),
  (Paper$2->Keyword$0.1)}
(Author -> Paper).S
{
  (Author$0->Keyword$0.4.0),
  (Author$1->Keyword$0.4.0),
  (Author$2->Keyword$0.4.0)}
Q
{
  (Author$0->Keyword$0.3.0),
  (Author$1->Keyword$0.1),
  (Author$2->Keyword$0.2.0)}
Q fun/div (Author -> Paper).S
{
  (Author$0->Keyword$0.0.75),
  (Author$1->Keyword$0.0.25),
  (Author$2->Keyword$0.0.5)}

```

Figure 7.5: Potential instance of the bibliographic database under $Mat_{\mathbb{R}_0^+}$.

a quick attempt at an adaptation of the current Alloy extension to real numbered matrices. Such solution still presents the other relations S , Q and so on, as intended while also being able to produce much more interesting Z s, modeling that relation as envisioned. Thus, this motivates that, in the future, other kinds of matrix, namely real numbered matrices, should be properly studied so that they can be implemented and further strengthen Alloy, expanding even more the range of problems that can be dealt with effectively through it.

7.2 FOOTBALL CHAMPIONSHIP

When performing quantitative analysis over the football tournament scheduler, its original properties preserve their meaning, resulting in instances like the one displayed in Figure 7.6, which represents a valid agenda for the number of teams and date slots specified, being also one of the solutions that can be found through SAT solving, containing ordinary relations only.

this/Game	home	away	date	s
Game ⁰	Team ¹	Team ²	Date ⁵	Game ²
Game ¹	Team ⁰	Team ²	Date ¹	Game ³
Game ²	Team ²	Team ¹	Date ⁰	Game ⁰
Game ³	Team ²	Team ⁰	Date ³	Game ¹
Game ⁴	Team ⁰	Team ¹	Date ⁴	Game ⁵
Game ⁵	Team ¹	Team ⁰	Date ²	Game ⁴

this/Date	Date ⁰	Date ¹	Date ²	Date ³	Date ⁴	Date ⁵

this/Team	Team ⁰	Team ¹	Team ²

Figure 7.6: An instance produced after quantitative solving the Football Championship specification from Appendix A.2.1.

However, with the increased expressiveness of the tool, now there are also new alternative ways of specifying properties, in particular, it is possible to define the so called **quantitative invariants** (Oliveira, 2017b).

For instance, recall the requirement a):

- The same team cannot play two games on the same day.

modeled as,

$$\begin{aligned}
& (away \cup home)^\circ \cdot (away \cup home) \cap date^\circ \cdot date \subseteq id \\
& \equiv \{ \text{Inclusion (2.6)} \} \\
& g'((away \cup home)^\circ \cdot (away \cup home) \cap date^\circ \cdot date)g \Rightarrow g' = id \ g \\
& \equiv \{ \text{Meet (2.7); Identity (2.2)} \} \\
& g'((away \cup home)^\circ \cdot (away \cup home))g \wedge g'(date^\circ \cdot date)g \Rightarrow g' = g \\
& \equiv \{ \text{Composition (2.3)} \} \\
& \langle \exists t : g' (away \cup home)^\circ t : t (away \cup home) g \rangle \wedge \\
& \langle \exists d : g' date^\circ d : d = date g \rangle \Rightarrow g' = g \\
& \equiv \{ \text{Converse (2.5); Join (2.8)} \} \\
& \langle \exists t : t = away g' \vee t = home g' : t = away g \vee t = home g \rangle \wedge
\end{aligned}$$

$$\begin{aligned}
& \langle \exists d : d = \text{date } g' : d = \text{date } g \rangle \Rightarrow g' = g \\
& \equiv \{ \text{Eindhoven One-point: } \langle k : k = e : T \rangle = T[k := e] \} \\
& \langle \exists t : t = \text{away } g' \vee t = \text{home } g' : t = \text{away } g \vee t = \text{home } g \rangle \wedge \\
& \text{date } g' = \text{date } g \Rightarrow g' = g
\end{aligned}$$

i.e., if a given team participates in two games (either at home or away) and both games are scheduled for the same date, then they represent the same match. Furthermore, it is encoded in Alloy as

$$(\text{away} + \text{home}).\sim(\text{away} + \text{home}) \ \& \ \text{date}.\sim\text{date} \ \text{in} \ \text{idem}$$

like presented in the Appendix A.2.1.

Turns out that this same invariant can be imposed in a simpler way by defining it under *Mat*, as follows (Oliveira, 2017b):

$$\begin{aligned}
& \text{date} \cdot (\text{away} + \text{home})^\circ \leq \top \\
& \equiv \{ \text{Going pointwise; Composition (2.25)} \} \\
& \langle \sum g :: d = \text{date } g \times g(\text{away} + \text{home})^\circ t \rangle \leq d \top t \\
& \equiv \{ \text{Transpose (2.5); Bilinearity of addition; By definition, } b \top a = 1 \} \\
& \langle \sum g :: d = \text{date } g \times t \text{ away } g + t \text{ home } g \rangle \leq 1 \\
& \equiv \{ \text{date is a function, therefore it is represented by a Boolean matrix;} \\
& \text{Eindhoven Trading: } \langle \sum k :: M \times N \rangle = \langle \sum k : M : N \rangle \text{ if } M \text{ is a Boolean matrix.} \} \\
& \langle \sum g : d = \text{date } g : t \text{ away } g + t \text{ home } g \rangle \leq 1
\end{aligned}$$

which counts for every date how many games are played by each team, and requires that it cannot exceed 1. Moreover, it can be specified using the extended Alloy language as:

$$\sim(\text{away} + \text{home}).\text{date} \leq \text{Team} \rightarrow \text{Date}$$

By replacing the initial invariant with the quantitative one, as presented in the Appendix A.2.3, Alloy is able to find instances which meet all the desired requirements, as expected. More, by specifying the quantitative invariant, not only the model becomes tidier, but the reduced complexity also potentially decreases the load on the solver, as the number of assertions generated on the respective SMT specification also goes down, easing the tool's job when enumerating different possible tournament schedules, for instance.

To make sure that both ways of encoding the same property are equivalent, one can assert the initial version and then check the assertion with varying scopes, similarly to the pictured in Figure 7.7, therefore being able to conclude with a certain degree of confidence that they act the same, as anticipated for this case in particular, especially after studying their respective pointwise correspondence. Consequently, the user has the freedom to decide between the two, generally picking the one which results in lower solving times in practice. To be noted that, since at this point the model is providing the same instances as the

```

assert oneGameDate{ (away + home) . ~(away + home) & date . ~date in iden }
check oneGameDate
for exactly
    3 Team,
    6 Game,
    6 Date

```

```

Executing "Check oneGameDate for exactly 3 Team, 6 Game, 6 Date"
Solver=CVC4 Integer Maximum weight=unlimited
145 primary variables, 2027 function symbols, 2045 assertions. 72ms.
No counterexample found. oneGameDate may be valid. 37961ms.
Elapsed time: 38035ms.

```

Figure 7.7: Checking if the alternative invariants associated with the requirement a) behave the same for a given set of participating teams and dates available.

qualitative counterpart, given that SAT solvers, in general, perform better in response time terms (naturally, since they work over a narrowed domain comparatively), Boolean analysis should be preferred, using the original version of the Alloy specification. However, once measurable data is added to the model, and qualitative analysis can no longer be performed, embracing this concept of quantitative invariants can help in optimizing the model at hand in the long term, potentially improving clarity, efficiency and so on.

Thus, the Alloy extension into the quantitative realm opens up the door to the improvement of new and existing Alloy models alike, by offering a new set of specification capabilities that arise from having its basis on *Mat*.

The next step is to study the model adapted to be able to track the tournament's results, through the relation $Result \xleftarrow{History} Team$ presented previously in Section 4.1.2, modeled using SMT in that case. Now, this quantitative relation can be encoded with the Alloy extension:

```

abstract sig Result{}
one sig Win, Lose extends Result{}

sig Team{ History : set Result }

```

At this point, this specification could still be subject to qualitative analysis, however it would only be able to reason over the reachability details associated with it, i.e., one would only be able to conclude if each team has won and lost games or if it either exclusively wins or loses matches. Nevertheless, by performing quantitative solving, the number of games that a team has won or lost can be precisely handled, but, without anything else Alloy will provide instances where the number of victories and defeats of each team vary freely within the \mathbb{N}_0 domain/integer scope limit, regardless of the tournament structure at hand, therefore further quantitative invariants must be imposed to properly model this relation.

In the previous SMT specification used to model the extended scenario during tool research, integer function symbols described the relation and constraints over them assured that the new requirements were met. In the same vein, the following quantitative constraints were imposed through the quantitative Alloy capabilities:

- There is a winner for every loser.

$$\sum Win \cdot History = \sum Lose \cdot History$$

- Every team either wins or loses each game in which they take part.

$$\langle \forall t :: \sum \underline{t} \cdot (home + away) = \sum History \cdot \underline{t} \rangle$$

encoded as,

```
#History.Win = #History.Lose
all t : Team | #(home + away).t = #t.History
```

These invariants allow for **explicit quantification**, i.e., they are constraints which take advantage of the quantitative semantics of the extended language, meaning that the usual qualitative analysis can no longer be executed over this model.¹ A possible Alloy instance for the quantitative specification is presented in Figure 7.8, which coincides precisely with the solution determined through the SMT model, shown previously

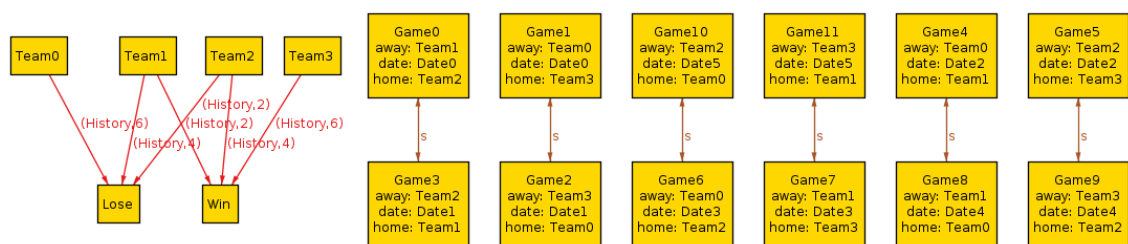


Figure 7.8: A solution to the quantitative Alloy specification of the Football Championship.

in the Table 4.1. Then, the resulting Alloy model (the full specification is presented in Appendix A.2.4) is able to properly determine agendas that abide to the tournament format as well as explicitly model the championship's results.

Finally, further additions to the model can be made following the same train of thought, allowing the manipulation over the results and, for example, model the point system over the football championship, being able to show the scoreboard, point out the winner team, the team in the last place, and others.

¹ Technically only the quantitative invariant presented for the requirement a) uses syntax which is not supported in the original Alloy language (comparison between relational expressions using inequality operators), so if the initial version of this requirement is used, SAT solvers *can* be used but, due to these invariants acting on the relation's weights, they will not be able to find any satisfiable instance with weights outside the $\{0, 1\}$ values.

Moreover, consider that a certain schedule determined by Alloy is being put into practice, then, the model can be updated after each match to include the new results and continuously be able to predict how many possible outcomes are left, which matches a given team has to win to reach first place, if a team is still able to win the championship, and so on.

7.3 SPRINKLER

Consider a shift in the target domain, from $\mathbb{F} = \mathbb{N}_0$ to $\mathbb{F} = [0, 1]$. Then, let us now assess how the extension behaves under a probabilistic setting, by attempting to model the Bayesian Network previously presented (Figure 4.2) under relational terms.

The state of whether it is raining or not, the sprinkler is currently on or off and the grass is wet or dry, can be modeled as $R = \{ Rain, NoRain \}$, $S = \{ On, Off \}$, $G = \{ Wet, Dry \}$, which in turn are encoded as Alloy *signatures*:

```
abstract sig R, S, G{}
```

```
one sig Rain, NoRain extends R{}
```

```
one sig On, Off extends S{}
```

```
one sig Wet, Dry extends G{}
```

Furthermore, every node present in the network will give origin to a relation, as expected from having previously reasoned over such structure algebraically in Chapter 4:

- $R \xleftarrow{\text{rain}} 1$
one sig Unit{ rain : one R }
- $S \xleftarrow{\text{sprinkler}} R$
abstract sig R{ sprinkler : one S }
- $G \xleftarrow{\text{grass}} S \xleftarrow{\quad} R$
abstract sig R{ grass : S set -> one G }

At last, the conditional probability table assigned to each node can be encoded through weight specification, as follows:

```
fact{
  // rain
  #(Unit.rain := Rain) = div[2, 10]
```

```

// sprinkler
#(NoRain.sprinkler :> On) = div[4, 10]
#(Rain.sprinkler :> On) = div[1, 100]

// grass
one Off.(NoRain.grass) :> Dry
#(Off.(Rain.grass) :> Dry) = div[2, 10]
#(On.(NoRain.grass) :> Dry) = div[1, 10]
#(On.(Rain.grass) :> Dry) = div[1, 100]
}

```

Note that it is not necessary to enforce all the entries of the tables exhaustively, since each relation will be represented by a left-stochastic matrix, by specifying $r - 1$ (where r is the number of rows in the table) entries, the last will be derived automatically since the sum of each column must add up to 1. After providing the specification (presented in Appendix A.3.3) to Alloy, the latter is able to determine an instance which properly represents this network, as pictured in Figure 7.9.

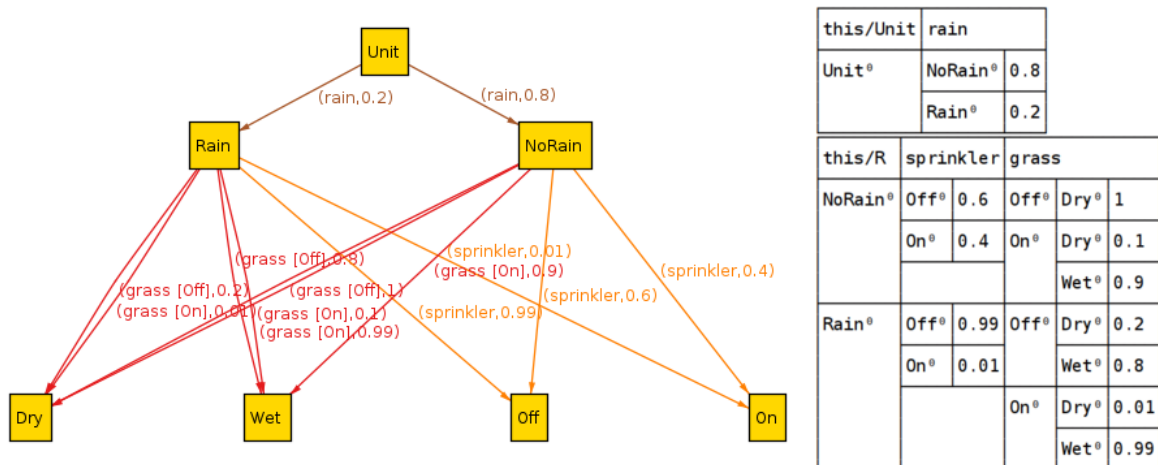


Figure 7.9: Alloy instance representing a Bayesian Network for the given *rain*, *sprinkler* and *grass*.

The previous solution is the only one for this specification, since all the relations within the model are known and constant. Then, the next step is to consider that a probabilistic table is unknown for, at least, one node of the network. When this network was previously modeled using PRISM, *sprinkler* was the object of study when checking if the latter was able to seek answers to questions like “Is there a *sprinkler* for which the probability of the grass being wet is higher than 90%?”, which was the case. Therefore, this section will follow that trend, to appraise if the quantitative component of Alloy is able to meet the same results. To specify an unknown *sprinkler*, the constraints previously imposed should simply be removed, so that

its weights can be freely assigned under the $[0, 1]$ domain; even partial conditional probability tables can be specified, and Alloy will attempt to fill out the rest with respect to the model's constraints. Figure 7.10 shows an instance provided after solving the model again without the *sprinkler's* weight constraints. At its

this/Unit	rain	
Unit ^o	NoRain ^o	0.8
	Rain ^o	0.2

this/R	sprinkler		grass		
NoRain ^o	Off ^o	0.3333333333333333	Off ^o	Dry ^o	1
	On ^o	0.6666666666666666	On ^o	Dry ^o	0.1
				Wet ^o	0.9
Rain ^o	Off ^o	0.1666666666666666	Off ^o	Dry ^o	0.2
	On ^o	0.8333333333333334		Wet ^o	0.8
			On ^o	Dry ^o	0.01
				Wet ^o	0.99

Figure 7.10: Example of a possible *sprinkler* found by Alloy.

current state, in contrast to the initial specification, there are infinite solutions to this model, in particular, every possible combination of $[0, 1] \times [0, 1]$, as there are virtually no other constraints imposed over the *sprinkler*. Thus, to extract further information from the model and make Alloy find interesting *sprinklers* that abide to certain properties, the Alloy specification shall be extended so that a response to the previous question “Is there a *sprinkler* for which the probability of the grass being wet is higher than 90%?” can be obtained, alongside being able to take advantage of the Bayes' theorem.

As mentioned before, the probability of the grass being wet can be measured as $P(g = 1) = wet \cdot grass \cdot (sprinkler \triangleright id) \cdot rain$. This expression cannot be immediately encoded in Alloy since Khatri-Rao product is not an operator supported in neither the original Alloy language nor the extension. Even though an attempt was made to define Khatri-Rao product in the quantitative Kodkod extension, the implementation accomplished was not very manageable nor useful to specify these kinds of expressions, and thus, it was not added to Alloy, since, from its definition $M \triangleright N : p \rightarrow m \times n$ (2.28), Khatri-Rao is used to define the *product* under the *Mat* setting, which does not cope well with both Kodkod and Alloy, since they only support “*curried*”-like relational types, despite $p \rightarrow m \times n \cong p \rightarrow m \rightarrow n$, there is no way of either “*currying*” or “*uncurrying*” the relations' type so that the Alloy operators, in this particular case, composition could be applied between *grass* and $(sprinkler \triangleright id)$, making it not as immediate to calculate $P(g = 1)$ using Alloy. The implementation of Khatri-Rao in Kodkod produces $M \triangleright N : p \rightarrow m \rightarrow n$ and thus, $(sprinkler \triangleright id) : R \rightarrow S \rightarrow R$. However, as $grass : R \rightarrow S \rightarrow G$, $grass \cdot (sprinkler \triangleright id) : R \rightarrow S \rightarrow S \rightarrow G$, which was not useful to model the intended expression.

Nonetheless, not being able to explicitly specify $A \times B$ as a single type is to be expected, given the matrix representation adopted in Kodkod, as well as the way atoms, tuples and the relations' bounds are specified,

otherwise it would require that $A \times B$ could be represented by a single column of the matrix (similar to the matrix representation of *grass* in Section 4.1.3) and then every possible $(a, b) \in A \times B$ would need to be represented similarly to an atom, thus, taking into account all the *signatures* present in the model as well as all possible combinations through the product, the size of the universe would explode, making it infeasible. In fact, it would actually be impossible to use the current finite representation to handle the product in such a way, since it would require to exhaustively enumerate all the infinite combinations, e.g., even a single atom A could be infinitely combined as $A \times A \times \dots$.

In this scenario, the solution to overcome this limitation would therefore be to model the necessary product as well as the application of Khatri-Rao product explicitly within the Alloy specification itself, that is, $S \times R$ must be represented by a single type and thus, encoded as a *signature*, and the model's relations then adapted to its inclusion:

- $S \times R$

```
abstract sig RS{ ... }
one sig RainOff, RainOn, NoRainOff, NoRainOn extends RS{}
```
- *grass* would then need to have its type adjusted from $R \rightarrow S \rightarrow G$ into $S \times R \rightarrow G$

```
abstract sig RS{ grass : one G }
```
- the constraints over *grass* must also be re-defined

```
one NoRainOff.grass :> Dry
#(RainOff.grass :> Dry) = div[2, 10]
#(NoRainOn.grass :> Dry) = div[1, 10]
#(RainOn.grass :> Dry) = div[1, 100]
```

At this point, this model behaves the same as the previous version. Now, to be able to model the use of Khatri-Rao, the helper relation $\text{spID} = \text{sprinkler} \triangleright \text{id}$ was also introduced to represent the result of its application:

```
abstract sig R{
  sprinkler : one S,
  spID : one RS
}
```

Lastly the Khatri-Rao product between *sprinkler* and *id* is modeled as:

```
#(Rain.spID :> RainOn) = #(Rain.sprinkler :> On)
#(Rain.spID :> RainOff) = #(Rain.sprinkler :> Off)
#(NoRain.spID :> NoRainOn) = #(NoRain.sprinkler :> On)
#(NoRain.spID :> NoRainOff) = #(NoRain.sprinkler :> Off)
```

It is worth mentioning that tuples like (Rain, NoRainOn) will never occur in spID due to it being represented by a left-stochastic matrix, as long as sprinkler is a proper probabilistic relation.

All the conditions needed to be able to encode $P(g = 1)$ in Alloy and further calculate its value have been met, and it can be done as follows:

```
fun grassWet : G { Unit.rain.spID.grass :=> Wet }
```

Then, by considering the previous *sprinkler*, the solution pictured in Figure 7.11 is obtained. The quantitative

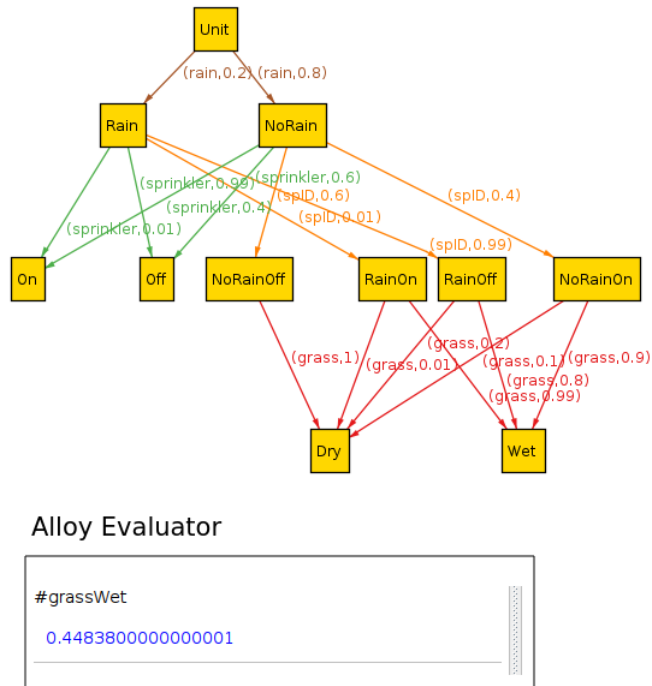


Figure 7.11: $P(g = 1)$ measured with respect to the presented network.

Alloy extension was able to successfully calculate $P(g = 1) = 44.838\%$, exactly the value obtained by hand using typed linear algebra and through PRISM in Section 4.1.3.

To finally be able to respond to the question on whether there is a *sprinkler* for which $P(g = 1) > 90\%$, one can assert that $P(g = 1) \leq 90\%$ and check if Alloy is able to find a counterexample. Figure 7.12 shows that Alloy was, in fact, capable of finding such a *sprinkler* = $\{(NoRain \rightarrow On, 1), (Rain \rightarrow On, 0.9913), (Rain \rightarrow Off, 0.0087)\}$ for which the probability is higher than 90%, $P(g = 1) = 91.77\%$. Thus, through the quantitative Alloy extension one is able to reason over Bayesian networks, measure probabilities, check for conditional probability tables which make a specified event happen with the desired chance, and so on.

For instance, conditional probabilities can also be specified and calculated, namely through Bayes' theorem:

$$P(A | B) = \frac{P(B | A)P(A)}{P(B)} \tag{7.1}$$

```
assert unknownSprinkler{ #grassWet <= div[9, 10] }
check unknownSprinkler
```

Executing "Check unknownSprinkler"

Solver=CVC4 Probabilistic Maximum weight=unlimited
 55 primary variables, 235 function symbols, 256 assertions. 124ms.
Counterexample found. unknownSprinkler is invalid. 77ms.
 Elapsed time: 267ms.

Alloy Visualizer

this/R	sprinkler		spID	
NoRain ^o	On ^o	1	NoRainOn ^o	1
Rain ^o	Off ^o	0.008670520231213872	RainOff ^o	0.008670520231213872
	On ^o	0.9913294797687862	RainOn ^o	0.9913294797687862

Alloy Evaluator

```
#grassWet
0.917670520231214
```

Figure 7.12: Finding a *sprinkler* for which $P(g = 1)$ is greater than 90%.

using Alloy like so:

FORWARDS REASONING $P(g = 1 \mid r = 1) = wet \cdot grass \cdot (sprinkler \vee id) \cdot raining$, where

$$R \xleftarrow{raining} 1 = [0 \ 1]^T$$

```
fun gwIfRain : G { Rain.spID.grass :> Wet }
```

BACKWARDS REASONING

$$P(r = 1 \mid g = 1) = \frac{P(g = 1 \mid r = 1)P(r = 1)}{P(g = 1)}$$

```
fun rainIfGW : Int {
  #gwIfRain fun/mul div[#(Unit.rain :> Rain), #grassWet]
}
```

Figure 7.13 presents the value of $P(g = 1 \mid r = 1)$ and $P(r = 1 \mid g = 1)$ calculated for the *rain*, *sprinkler* and *grass* initially considered. So, $P(g = 1 \mid r = 1) = 80.19\%$ and $P(r = 1 \mid g = 1) = 35.77\%$, as expected from (Oliveira, 2017a).

PROBABILISTIC CONTRACT Conditional probability $P(a \mid b)$ considers the probability of an event a occurring taking into account the assumption that event b happens. This line of thought points towards the notion of probabilistic contract, with b corresponding to the precondition and a acting as the postcondition for which the contract $a \xleftarrow{f} b$ will be evaluated over a probabilistic function f describing the system

```
#gwlfRain
0.8019000000000001
#rainIfGW
0.3576876756322761
```

Figure 7.13: Calculating conditional probabilities through Bayes' theorem in Alloy.

being studied. Then, can the quantitative extension of Alloy also be used to handle this case study and determine, for instance, $P(g = 1 \mid r = 1)$ through probabilistic contract concepts?

As seen before, $1 \xleftarrow{wet} G = [0 \ 1]$ and $R \xleftarrow{raining} 1 = [0 \ 1]^T$ are both Boolean vectors representing $g = 1$ and $r = 1$ respectively, meaning that they are suitable precondition and postcondition for the contract $wet \xleftarrow{f} raining$ which will specify $P(g = 1 \mid r = 1)$ where $f : R \rightarrow S \times G$ is the probabilistic function that models this Bayesian Network's behaviour with respect to the probability of raining (Oliveira, 2017a):

$$f = sprinkler \triangleright (grass \cdot (sprinkler \triangleright id))$$

Once again, to be able to represent the Khatri-Rao product defining f , the product $S \times G$ must also be explicitly specified:

```
abstract sig SG{}
one sig OffDry, OffWet, OnDry, OnWet extends SG{}
```

Afterwards, the Alloy model is extended to include the field f , declared as:

```
abstract sig R{ f : one SG }
```

Finally, the definition of f is achieved by enforcing the necessary constraints to model the Khatri-Rao product:

```
let gis = spID.grass {
#(NoRain.f :=> OffDry) = mul[#(NoRain.sprinkler:=>Off),#(NoRain.gis:=>Dry)]
#(NoRain.f :=> OffWet) = mul[#(NoRain.sprinkler:=>Off),#(NoRain.gis:=>Wet)]
#(NoRain.f :=> OnDry) = mul[#(NoRain.sprinkler :=> On),#(NoRain.gis :=> Dry)]
#(NoRain.f :=> OnWet) = mul[#(NoRain.sprinkler :=> On),#(NoRain.gis :=> Wet)]
#(Rain.f :=> OffDry) = mul[#(Rain.sprinkler :=> Off),#(Rain.gis :=> Dry)]
#(Rain.f :=> OffWet) = mul[#(Rain.sprinkler :=> Off),#(Rain.gis :=> Wet)]
#(Rain.f :=> OnDry) = mul[#(Rain.sprinkler :=> On),#(Rain.gis :=> Dry)]
#(Rain.f :=> OnWet) = mul[#(Rain.sprinkler :=> On),#(Rain.gis :=> Wet)]
}
```

Now, to measure the contract $\llbracket wet \xleftarrow{f} raining \rrbracket_\delta$, the input distribution $\delta : 1 \rightarrow R$ is also included in the specification:

one sig Unit{ delta : one R }

and thus, the probability of the contract holding is calculated by (2.41):

$$\llbracket wet \xleftarrow{f} raining \rrbracket_{\delta} = (wet \cdot f \times raining) \cdot \frac{\delta}{raining \cdot \delta}$$

which in turn is encoded using Alloy syntax:

```
fun grass_wet : SG{ OffWet + OnWet }
```

```
fun raining : R{ Rain }
```

```
fun measureContract : Unit{
```

```
  delta.(f.grass_wet fun/mul raining) fun/div delta.raining
```

```
}
```

Solving the model (presented in Appendix A.3.5) for the constant *rain*, *sprinkler* and *grass* initially considered, Alloy produces an instance like the one presented in Figure 7.14. For this solution $\delta = [0 \ 1]^T$

this/R	sprinkler	spID	f	this/Unit	rain	delta
NoRain ^o	Off ^o	0.6	NoRainOff ^o	0.6	OffDry ^o	0.384
	On ^o	0.4	NoRainOn ^o	0.4	OffWet ^o	0.216
				OnDry ^o	0.256	
				OnWet ^o	0.144	
Rain ^o	Off ^o	0.99	RainOff ^o	0.99	OffDry ^o	0.196119
	On ^o	0.01	RainOn ^o	0.01	OffWet ^o	0.793881
				OnDry ^o	0.001981	
				OnWet ^o	0.008019	
				this/RS	grass	
				NoRainOff ^o	Dry ^o	1
				NoRainOn ^o	Dry ^o	0.1
					Wet ^o	0.9
				RainOff ^o	Dry ^o	0.2
					Wet ^o	0.8
				RainOn ^o	Dry ^o	0.01
					Wet ^o	0.99

Alloy Evaluator

```
#measureContract
0.8019
```

Figure 7.14: Measuring $P(g = 1 | r = 1)$ using probabilistic contracts in Alloy.

and $\llbracket wet \xleftarrow{f} raining \rrbracket_{\delta} = 80.19\% = P(g = 1 | r = 1)$, coinciding with the value arrived at previously, as desired.

However, one may now wonder, what is the value of $P(g = 1 | r = 1)$ for other possible interpretations of δ ? Is it independent of any input distribution, that is, no matter the δ considered, does the probability remain the same, as would be expected from the previous measurements? To ascertain if that is the case, one can consider the check command displayed in Figure 7.15. Alloy was unable to find any δ where the

```

assert contract{
  #measureContract = div[8019, 10000]
}
check contract

```

Executing "Check contract"
 Solver=CVC4 Probabilistic Maximum weight=1
 47 primary variables, 372 function symbols, 393 assertions. 28ms.
 No counterexample found. contract may be valid. 85ms.
Elapsed time: 116ms.

Figure 7.15: Check if there is a δ for which $P(g = 1 \mid r = 1) \neq 80.19\%$.

probability is different, meaning that as long as the probability of raining is non-zero (since if $P(r = 1) = 0$ then the probability of the contract holding is undefined due to division by zero), for this specific *grass* and *sprinkler*, the chance of the grass being wet given that it is raining is always the same, as intended. Do note that during this analysis the value of *rain* was irrelevant, as the assumption is that it will be raining. In the calculations made before, $rain = [0.80 \ 0.20]^T$ and since the probability of raining in this case is different from zero ($P(r = 1) = 20\%$), Alloy arrived at the same conclusion.

In the end, it was possible to model a Bayesian Network in Alloy, and further study its behaviour with respect to different circumstances, varying conditional probability tables, calculating probabilities through known results and even by taking advantage of probabilistic contracts.

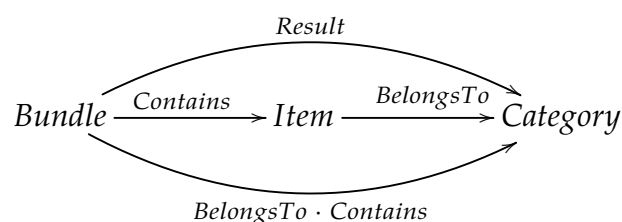
7.4 BUNDLING

The earlier Section 3.2.1 studied the index-based approach taken to implement *multirelations* in Alloy, developed by Sun et al. (2016) as an Alloy library. In their paper, they address the bundling process for the upcoming seasonal sales of a grocery store by taking advantage of *multirelations*.

Each bundle to be prepared for the sales must abide to the following requirements:

- (a) It must contain at least two food items.
- (b) At least two items within the bundle must be dairy products.
- (c) Variety is expected, at least two products in the bundle have to belong to different categories.

Such scenario can then be modeled through abstract diagram notation:



composed by the following relations:

- $c = \text{BelongsTo } i$ – Each food item i belongs to a single category c .
- $i \text{ Contains } b$ – Bundle b contains the product i .
- $c \text{ Result } b = c (\text{BelongsTo} \cdot \text{Contains}) b$ – Bundle b has food items of category c .

subject to the constraints:

- Every bundle must contain, at least, two dairy products – handling requirements a) and b).
 $\text{all } b : \text{Bundle} \mid \#(b <: \text{Result } :> \text{Dairy}) \geq 2$
- Each bundle has products from more than one category – requirement c).²
 $\text{all } b : \text{Bundle} \mid \#\text{drop}(b.\text{Result}) \geq 2$

The Alloy specification proposed by Sun et al. (2016) for a specific scenario of this problem is presented in Appendix A.5.1. *Contains* is modeled as a *multirelation*, meaning that $i \text{ Contains } b = n$ means that the bundle b includes exactly n units of item i . Furthermore, *Result* is also represented by a *multirelation*, being able to reason about **how many** items from each category are within each bundle. Since *Result* is described by the *multirelation composition* between the *multirelation Contains* and the ordinary relation *belongsTo*, then the latter must first be lifted into the *MRel* domain, that is, $\text{BelongsTo} = \text{lift belongsTo}$.

Considering $\text{Bundle} = \{B_1, B_2\}$, $\text{Item} = \{\text{Bread}, \text{Milk}, \text{Butter}\}$ and $\text{Category} = \{\text{Bakery}, \text{Dairy}\}$, the model shown in Appendix A.5.1, executed using SAT solvers, provides instances like the one presented in Figure 7.16. In this example, B_1 contains two units of *Butter* (consequently, two dairy

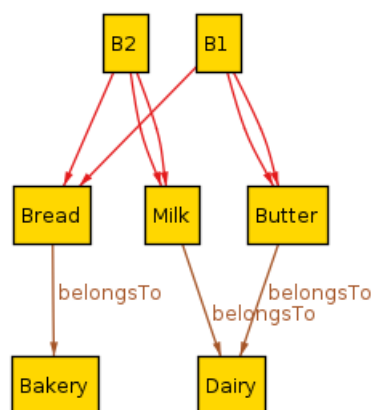


Figure 7.16: SAT solver solution to the Bundling problem using *multirelations*.

products) and one *Bread*; B_2 has one *Bread* and two units of *Milk*, being a dairy food item; thus both bundles obey to the initial rules.

² For simplicity, `d drop` refers to either the operator added to the Alloy extension or to the operation provided by the *multirelations* library depending on the context.

This problem can also be adequately handled using the quantitative extension of Alloy developed, under $\mathbb{F} = \mathbb{N}_0$, by making a few changes to the specification:

- There is no longer the need to instantiate explicitly which relations must be lifted to *multirelations*, as every relation will be represented by a matrix of $Mat_{\mathbb{N}_0}$. As already seen in previous examples, through the use of *drop* or multiplicity constraints, then the relations that need to be represented by Boolean matrices can be properly constrained to do so. In this case, only *BelongsTo* is an ordinary relation, in particular, a function, and thus it can be properly defined by taking advantage of the one declaration constraint:

```
abstract sig Item{ BelongsTo : one Category }
```

- *Contains* and *Result* are both declared as a *field* of *Bundle* instead of through the *multirelations* module.

```
abstract sig Bundle{
  Contains : set Item,
  Result   : set Category
}
```

- *Result* is defined with the usual composition instead of using *multijoin* and, therefore, *BelongsTo* does not need to be lifted before composing:

```
Result = Contains . BelongsTo
```

The full Alloy specification is presented in Appendix A.5.2. Figure 7.17 displays an example of a solution

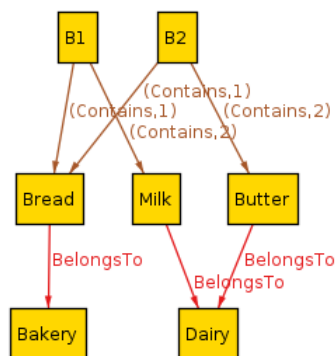


Figure 7.17: Example of an instance provided by Quantitative Alloy for the Bundling model.

provided for the adapted specification by the Alloy extension. Coincidentally, this instance is very similar to the previous example, with B_1 containing *Milk* instead of *Butter* and vice versa, meaning that it also abides to the desired properties.

Both specifications analysed can be adjusted to provide instances for more general values of *Bundle*, *Item* and *Category*. In particular, the property that demands that each bundle must be composed by at

least two dairy products was also generalized for *some* category instead of dairy especially:

$$\text{some } c : \text{Category} \mid \text{all } b : \text{Bundle} \mid \#(b <: \text{Result } :=> c) \geq 2$$

Figure 7.18 shows an instance of the adapted specification presented in Appendix A.5.3 using the *multirelations* concepts. Quantitative Alloy finds solutions like the one displayed in Figure 7.19 for the Alloy model in

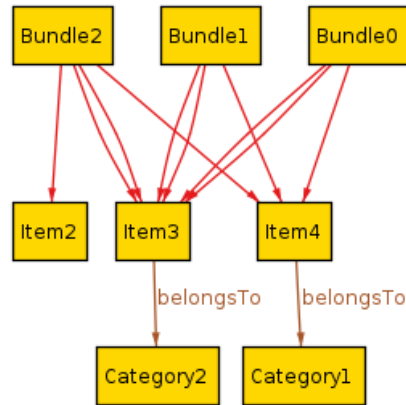


Figure 7.18: *multirelations* instance to the generalized bundling model.

Appendix A.5.4. This generalization of the specification will be taken advantage of in the next section.

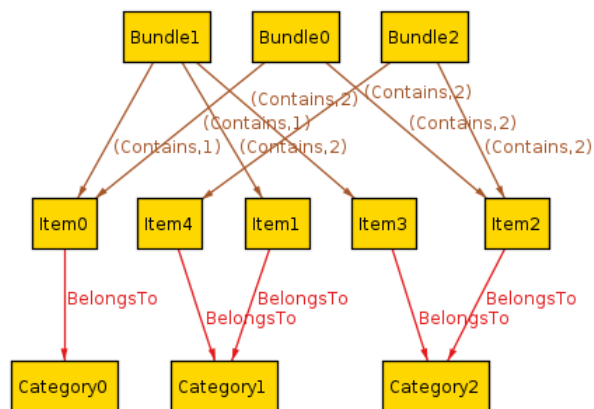


Figure 7.19: Solution to the general bundling specification provided by the quantitative extension.

Thus, Quantitative Alloy is also able to deal with such kind of models, analogously to the *multirelations* framework.

7.5 PERFORMANCE RESULTS

To appraise the performance of the extension developed in practice, this section goes over a simple benchmark made over the case studies presented previously when tackled using the original Alloy or the quantitative extension implemented.

SETUP The benchmark environment considered is characterized as follows: each test was executed locally in a machine equipped with 8GB of RAM and an octa-core Intel i7 CPU of 2.5GHz frequency and x86_64 architecture; the Alloy Analyzer ran with 768 MB of maximum memory and 8192k of maximum stack size; specifications under the Boolean domain will be solved using the SAT4J SAT solver with *skolem depth* = 1; the CVC4 SMT Solver will be used to handle the models under both integer and probabilistic quantitative analysis contexts with default settings.

Appendix A.6 presents exhaustively the response times obtained for each scenario considered for this benchmark, which will now be studied in greater detail.

BIBLIOMETRICS Starting off with the bibliometrics case study, the interest is to compare how SAT solvers and SMT solvers differ when given the same Alloy model (Appendix A.1.1). Afterwards, the quantitative adaptation (Appendix A.1.2) to the initial model, so that implicit quantification can be properly performed, is also subject to quantitative solving.

		Bibliometrics		
		Paper = 3 Author = 3 Medium = 1 Keyword = 1	Paper = 10 Author = 3 Medium = 2 Keyword = 3 Int = 6	Paper = 15 Author = 6 Medium = 3 Keyword = 5 Int = 6
SAT	variables	451	3246	9113
	primary variables	48	279	720
	clauses	784	7706	21460
	solving time	9.5 ms	59 ms	254.33 ms
		Paper = 3 Author = 3 Medium = 1 Keyword = 1	Paper = 10 Author = 3 Medium = 2 Keyword = 3	Paper = 15 Author = 6 Medium = 3 Keyword = 5
SMT	function symbols	311	1812	4805
	primary variables	38	237	569
	assertions	333	1834	4827
	solving time	116.67 ms	5.9 s	24.2 s
SMT <i>quantitative adaptation</i>	function symbols	417	2730	7467
	primary variables	41	246	599
	assertions	446	2759	7496
	solving time	114 ms	5.7 s	55.62 s

Table 7.1: Bibliometrics – SAT versus SMT.

Naturally, using SAT solvers result in faster solving times, as shown in Table 7.1, which increases once the measurable data is added to the instances. Although the two kinds of analysis are being performed over the same model, these results are intended to highlight the contrast between them, not to act as a proper comparison, as they are working towards instances from completely distinct universes.

While the difference between the response times obtained using SMT Solvers was minimal, the changes made to the model using quantitative operators increased the problem size, making it more noticeable once the scope increases significantly.

All in all, even though the response times of SMT solvers are higher, which is to be expected given the wider domain in which the analysis is being performed, they are able to reach a solution in workable times.

FOOTBALL CHAMPIONSHIP Initially, the original setting for the scheduling of a football tournament is considered (specification in Appendix A.2.1), solved with both SAT and SMT Solvers as presented in Table 7.2.

		Football Championship		
		Team = 2 Game \leq 2 Date \leq 2	Team = 3 Game \leq 6 Date \leq 6	Team = 4 Game \leq 12 Date \leq 6
SAT	variables	213	1798	6380
	primary variables	20	120	330
	clauses	336	3246	12333
	solving time	8.5 ms	19.33 ms	81 ms
SMT	function symbols	272	1883	7110
	primary variables	22	123	334
	assertions	288	1899	7126
	solving time	> 7 minutes \rightarrow unknown	6 s	2.17 minutes

Table 7.2: Football Championship – SAT versus SMT.

As expected, when handling the same kind of instances with both SAT and SMT solvers, the first are able to reach an answer much faster. That is why, unless the increased expressiveness is required, performing the usual Boolean analysis using SAT solvers should still be preferred whenever possible.

Afterwards, the previous conjecture that the re-definition of the model's constraints using *quantitative invariants* can help reduce the load on the solver, and consequently also help reaching a solution faster, is put to test and the results of solving the adapted specification (see Appendix A.2.3) are displayed in Table 7.3.

		Football Championship – Quantitative Invariant		
		Team = 2 Game \leq 2 Date \leq 2	Team = 3 Game \leq 6 Date \leq 6	Team = 4 Game \leq 12 Date \leq 6
SMT	function symbols	248	1547	5478
	primary variables	22	123	334
	assertions	264	1563	5494
	solving time	144.33 ms	3.9 s	56.2 s

Table 7.3: Quantitative adaptation of the Football Championship constraints.

And the hypothesis checks out! The number of function symbols and assertions in the assertion stack were significantly reduced and the response times were almost cut in half; and, while with the initial version the solver was struggling to find an instance for smaller scopes, now it was able to arrive at the desired solution almost instantly.

Lastly, the benchmark evaluates the quantitative extension to the problem, enhanced with the *History* relation. To establish a comparison, the same setting was modeled by taking advantage of the *multirelations* library, encoding *History* as a *multirelation*, presented in Appendix A.2.5. Using SAT solvers together with the *multirelations* concepts, analogous instances as the ones obtained through Quantitative Alloy can be determined, as exemplified in Figure 7.20.

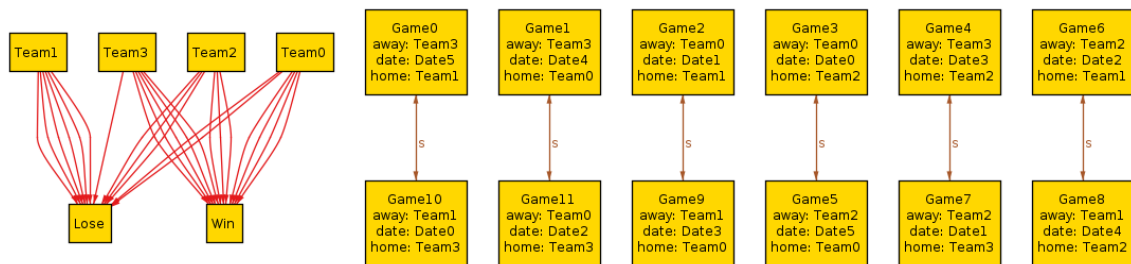


Figure 7.20: An instance of the Football Championship case study using *multirelations*.

Table 7.4 then presents the statistics associated with solving the augmented scenario using Quantitative Alloy (the model in Appendix A.2.4) and the previous specification taking advantage of SAT solvers.

		Football Championship with History		
		Team = 2 Game ≤ 2 Date ≤ 2	Team = 3 Game ≤ 6 Date ≤ 6	Team = 4 Game ≤ 12 Date ≤ 6
SMT	function symbols	291	1620	5597
	primary variables	28	131	344
	assertions	312	1641	5618
	solving time	260 ms	37 s	55.88 s
		Team = 2 Game ≤ 2 Date ≤ 2 History/Head ≤ 4 Int = 4	Team = 3 Game ≤ 6 Date ≤ 6 History/Head ≤ 12 Int = 5	Team = 4 Game ≤ 12 Date ≤ 6 History/Head ≤ 24 Int = 6
SAT <i>multirelations</i>	variables	550	3651	11147
	primary variables	40	192	498
	clauses	1318	9305	29033
	solving time	13.5 ms	77 ms	650.5 ms

Table 7.4: Football Championship with the quantitative relation *History*.

Turns out that using the *multirelations* is advantageous for the scopes considered, which is not surprising, as when this approach was studied previously, together with the power of SAT solvers, it was seen that it is very effective to handle problems with a reduced number of quantitative relations and/or with low integer values, that is, not requiring a very large integer scope to achieve the desired solutions, as it depends directly on the maximum number of occurrences that the same tuple can occur within a given *multirelation*, in this case, the scope of *History/Head* must be fixed, which is required to be at least high enough to represent the maximum number of times that a team can either win or lose, as well as requiring an integer scope as high or even higher, so that measuring the value of the number of edges of some tuples and operations between them can be evaluated into accurate results, that is, avoiding integer overflow. Nevertheless, even

though the response times were slower using Quantitative Alloy, they are still admissible and usable in practice. But given the choice, the user has the freedom to decide which approach should be preferred taking the circumstances of the problem at hand into account.

BUNDLING The last example seen under the domain of integer values is the bundling model.

In a first phase, the original commands considered in (Sun et al., 2016) were studied, with fixed values of *Bundle*, *Item*, *Category* and *BelongsTo*. This scenario, specified in the Appendixes A.5.1 and A.5.2, was solved using the SAT and SMT solvers respectively, whose results are presented in Table 7.5.

		Bundling	
		Contains/Head = 6 BelongsTo/Head = 3 Result/Head = 20	check Contains/Head = 6 BelongsTo/Head = 3 Result/Head = 20
SAT <i>multirelations</i>	variables	6583	6709
	primary variables	340	342
	clauses	15821	15966
	solving time	116.33 ms	359.83 ms
		for 10	check for 10 int
SMT	function symbols	150	159
	primary variables	23	25
	assertions	159	197
	solving time	39.5 ms	37.5 ms

Table 7.5: Bundling example performance.

In this case, the quantitative Alloy extension performs better than using SAT solvers. Furthermore, in the case of the *check* command, it has the advantage of reaching a conclusion about the whole \mathbb{N}_0 domain associated with the weights, with respect to the scope of the universe only, while in the *multirelations* case, the number of possible arcs is upper bounded by the number of *heads* of the spans imposed in the scope, in this case, for example, 6 *Contains/Head* means that only instances where the weight of each tuple within the relation *Contains* does not exceed 6 are considered.

Next, this example is further studied in more general terms, solving the Alloy specifications presented in Appendixes A.5.3 and A.5.4 instead, having obtained the data displayed in Table 7.6.

One of the downsides of the *multirelations* library is that its performance is reliant on a low number of *head* atoms in the universe, that is, a low value of tuple weights. Solving this model over greater scopes requires a greater number of *heads* to properly be able to reach valid instances as well as an integer scope high enough to handle the number of arcs of each relation, exploding the problem size and thus, taking too big of a tool on the SAT solver, making it unusable in practice.

On the other hand, the quantitative extension is not impacted nearly as heavily when given greater problem sizes, by not requiring such big leaps on the scope, proving to be more scalable, with the added advantage of also working over unbounded non-negative integers by default.

		Bundling – Generalized		
		Bundle = 3 Item = 5 Category = 3 Contains/Head = 10 BelongsTo/Head = 10 Result/Head = 10 Int = 6	Bundle = 5 Item = 10 Category = 3 Contains/Head = 30 BelongsTo/Head = 30 Result/Head = 30 Int = 6	Bundle = 10 Item = 20 Category = 5 Contains/Head = 40 BelongsTo/Head = 40 Result/Head = 50 Int = 8
SAT <i>multirelations</i>	variables	14703	193751	591348
	primary variables	468	3003	7185
	clauses	39662	580468	1850412
	solving time	5.5 s	> 35 minutes without response	–
		Bundle = 3 Item = 5 Category = 3	Bundle = 5 Item = 10 Category = 3	Bundle = 10 Item = 20 Category = 5
SMT	function symbols	353	779	2978
	primary variables	53	116	390
	assertions	357	779	2968
	solving time	139.17 ms	453.17 ms	6.1 s

Table 7.6: Stats of solving the generalized bundling example.

SPRINKLER Moving on to the probabilistic context, this benchmark evaluates how well Alloy is able to find unknown *sprinklers* of the Bayesian Network considered with respect to some property, in this case, depending on the probability of the grass being wet. Table 7.7 contains the details obtained after solving the model presented in Appendix A.3.4 with respect to each property.

	Bayesian Network – Unknown sprinkler			
	any sprinkler	$P(g = 1) = 50\%$	$P(g = 1) \geq 90\%$	$P(g = 1) < 20\%$
function symbols	185	204	204	204
primary variables	33	33	33	33
assertions	203	222	222	222
solving time	44.33 ms	42 ms	48.17 ms	49.83 ms

Table 7.7: Finding different *sprinklers*.

Alloy is able to determine *sprinklers* which obey to the constraints imposed rather quickly.

On a similar note, the previous example considered in Section 6.5 to showcase probabilistic contracts was also analysed, to see how the tool responds when attempting to find an input distribution that either serves as a counterexample for a check command or that makes the contract hold with a desired probability. Then, after taking the specification in Appendix A.4, Alloy was able to find δ s in the timeframe presented in Table 7.8.

	Measure contract $\{b_2\} \xleftarrow{f} \{a_1, a_2\}$			
	check $\geq 90\%$	run = 78.3%	run $> 78.3\%$	run $< 78.3\%$
function symbols	89	86	86	86
primary variables	15	15	15	15
assertions	95	93	93	93
solving time	23.33 ms	24.33 ms	25 ms	24.33 ms

Table 7.8: Testing Alloy when finding δ s for the probabilistic contract considered.

Like when finding *sprinklers*, the tool arrives at suitable δ s in very low response times.

Both probabilistic objects of study result in small Alloy models structurally, as their main interest is precisely about the properties that arise from the weights of the relations' tuples. Thus, taking these results and the ones from the previous case studies into account, one can come to the conclusion that the performance of the quantitative extension depends more on the universe size, the scope, rather than the intricate numeric values associated, being able to effectively calculate the values that meet the quantitative properties imposed.

In the end, through this benchmark one is able to come to the conclusion that Quantitative Alloy pulls up its own weight against the competition, being practical and performing the solving process in reasonable amounts of time or even faster than the comparisons, for all the problems considered. Moreover, these results suggest a rule of thumb when choosing between which kind of analysis to be executed:

1. If SAT solvers possesses enough expressiveness to solve the model, then the user should prefer Boolean analysis.
2. Instead, if the model contains quantitative information and thus, SAT solvers cannot be used:
 - 2.1. $\mathbb{F} = \mathbb{N}_0$, does the model display a reduced amount of quantitative relations and a bounded range of integers suffices to properly analyse it? Then, if the *multirelations* library can be used to model the problem, applying it and solving the specification using SAT solvers should be preferred.
 - 2.2. For probabilistic settings or integer problems in general or of greater size, which include multiple quantitative relations and/or require a greater integer range or even unbounded integers to verify the desired properties, then the quantitative extension of Alloy will perform the best.

7.6 SUMMARY

Throughout this chapter we recall the case studies previously presented and attempt to solve them using the quantitative extension of Alloy implemented:

- It extracts bibliometrics from the already existing model of a bibliographic system;
- Within the football tournament scenario, suitable agendas are found using quantitative solving; a requirement is modeled in a cleaner way through the quantitative capabilities of the extension; the quantitative relation *History* is specified using Alloy and further instances of the championship schedule, together with the number of victories and defeats of each team, are found;
- The Bayesian Network is successfully encoded as an Alloy model, which is further used to calculate interesting probabilities both by algebraically means and through probabilistic results like the Bayes'

theorem; use the tool to find conditional probability tables for certain nodes that meet some desired properties; model and measure a probabilistic contract over this network to calculate the probability of a certain property.

Furthermore, the enhanced tool is used to also handle the seasonal sales example presented in (Sun et al., 2016), modeled before by the authors using *multirelations* concepts.

Finally, the chapter ends with a benchmark evaluating the performance of the extension compared to the original Alloy and against the related work studied in Section 3.2.1, as well as how it behaves on its own when handling domains for which neither of the previous can be applied.

CONCLUSION

This final chapter looks back and evaluates the progress made over the initial aim to perform quantitative solving using Alloy, the groundwork laid to accomplish this and the actual Quantitative Alloy extension developed. Such conclusions are followed by a prospect of future work on what needs to be done to push these ideas forward.

8.1 CONCLUSIONS

Enriched with the quantitative extension proposed and implemented in this dissertation, Alloy is not only capable of providing answers to satisfiability problems within the usual Boolean setting, but also over inherently quantitative scenarios, now offering capabilities to analyse measurable information. By being able to perform quantitative solving of Alloy specifications on demand, the tool becomes applicable to problems of the kind addressed in so-called *quantitative formal method*.

Taking *Category Theory* and *Typed Linear Algebra* to form the mathematical framework of this dissertation proved to be a good choice, as both blend well with the Alloy characteristics, while at the same time being capable of bringing out the quantitative side into the original design. By shifting from *Rel* to *Mat*, $+$ -idempotency was effectively avoided, and thus the main obstacle of quantification using Alloy was removed.

As shown in Chapter 2, expressing relations and relational concepts through typed linear algebra allowed for quantitative reasoning seamlessly, under both categories of matrices composed by natural numbers $Mat_{\mathbb{N}_0}$ and left-stochastic matrices LS as well. Such a theory was successfully put into practice in Chapter 4, where it was applied to the *Bibliometrics* (with $Mat_{\mathbb{N}_0}$) and *Sprinkler* (using LS) case studies.

The increase of expressiveness implicit in the planned quantitative Alloy extension led to several design choices concerning extending components of Kodkod and Alloy. Chapter 5 addresses the most important ones. By studying different examples and case studies, confidence increased about such decisions. In particular, model specification and further analysis was accomplished (as seen in Chapter 7) in a way resembling that of standard Alloy, potentially easing the learning curve that Alloy users might experience when attempting to perform quantitative analysis through the proposed Alloy extension.

In order to operate over the new kinds of values, Kodkod requires equally expressive structures to support them. For that, alongside with numeric matrices, Numeric Circuits were established to reason over the numeric values and operations/formulas over them. These circuits arose from the existing Compact Boolean Circuits, and proved to be adequate. In particular, they are able to handle CBCs, that is, help the quantitative solving of Kodkod problems that arise from existing Alloy models and, at the same time, include new kinds of values and operators to represent quantitative expressions.

Unfortunately, after shifting into the quantitative realm, previously existing optimization techniques over Boolean problems and CBCs are no longer fully applicable. In particular, translation-level optimizations like caching mechanisms during the assembly process is achieved for NCs as well, but *symmetry break* and *skolemizing* techniques, which act at the circuit level and, consequently, at the level of the specification that will be generated to be solved by the suitable solver, are not.

The latter are used to optimize both the main formula and bounds of the Kodkod problem before being translated into a CBC. Specifically, *symmetry break* first detects the problem's symmetries and then includes a predicate within the circuit to eliminate those symmetries to make the solver avoid finding isomorphic solutions at a structural level. Simply put, given a solution, further solutions that can be transformed into the previous by swapping the atoms/tuples around are ignored. Also, as mentioned when the Numeric Circuits were introduced, currently their assembly process does not ensure partial canonicity, meaning that shared components are not being actively detected and further optimized, like in the Kodkod Boolean analysis procedure.

With the help of the numeric structures implemented, nearly every part of the Kodkod abstract syntax was successfully characterized under quantitative semantics, save for the (reflexive) transitive closure operator, which is currently implemented analogously to its qualitative counterpart, by the matrix addition of an incremental number of compositions of the relation R at hand $\hat{M} = M + M \cdot M + M \cdot M \cdot M + \dots$, with $M = \llbracket R \rrbracket$, through iterative squaring, which does not necessarily work on the quantitative domain.

Moreover, the Kodkod syntax was successfully augmented with the new operations to aid with the quantitative specification process, taking advantage of the acquired expressiveness, like the addition of *drop*, relational comparison using inequality operators, and so on.

Having built a Numeric Circuit from the given Kodkod problem, the next step is to construct an adequate specification from it, to be later fed into the correspondent quantitative solver.

SMT SOLVER Generating an equivalent SMT2-LIB specification from the NC considered was achieved seamlessly, which in turn is solved with respect to the *theory of integers* (logic QF_NIA), possessing enough expressiveness to pass the judgement of the circuit encoded within the desired domain.

SMT Solvers perform exceptionally, being able to handle the $Mat_{\mathbb{N}_0}$ and arrive at a response in usable times (as seen in the benchmark presented in Section 7.5), synergizing with the Alloy nature, with which one is able to find solutions on demand. It is capable of incremental solving, allowing for efficient solution enumeration, as desired, since Alloy provides the user with the capability of iterating

over solutions freely. While CVC4 is the only SMT Solver currently integrated, it was implemented in a flexible way so that more SMT Solvers can be easily supported in the future.

In the initial plans SMT solving was going to be used only when handling systems over natural numbers. However, given how well it conformed to both the theoretical framework and the Numeric Circuits, it exceeded the initial expectations and was able to deal with probabilistic scenarios as well, simply by taking the *theory of reals* (fragment QF_NRA) as the background theory, and bounding the primary variables to the $[0, 1]$ interval.

In conclusion, SMT Solvers can be effectively used under both quantitative domains considered.

PRISM Sadly, PRISM does not hit the mark nearly as well as SMT solvers do.

Deriving a PRISM model from a NC goes well, as the language possesses enough expressiveness to encode the kind of expressions considered. Problems arise once the model needs to be solved. It turns out that PRISM does not cope well to the model finding nature of Alloy. In the translation implemented, every primary variable gives origin to a PRISM constant representing a probability and there lies the problem: while the interest is to calculate such probabilities, PRISM itself does not model check models with unknown constants, even though it provides mechanisms to work around it, and that is where the `SolvingModes` presented in Section 6.1 come into play:

- Experiments allow the user to define a range of possible values that each unknown constant can assume with respect to a certain step, meaning that every possible combination of the values that each primary variable can be assigned will be taken as a possible solution, which will be subject to an instance of model checking. Therefore, if the problem is composed by V primary variables, each with the same number of possible values N , the experiment will create a total of N^V model checking instances! Growing exponentially on the number of primary variables is unfeasible: even in the previous benchmark (Section 7.5), depending on the scope, the number of primary variables ranged between 15 and 599. Even if one assumes a “big” step, e.g., 0.1, and if every variable can range over $[0, 1]$, in the smallest example 11^{15} instances of model checking are created, which is an unbearable amount already, making it unusable in practice.
- Parametric Model Checking would be ideal but, unfortunately, at the time of writing this technique only supports very specific kinds of PRISM models. Moreover, while the model developed in Section 4.1.3 could be subject to PMC, the model generated automatically from the NC does not meet the conditions to do so.
- The last option, and the one adopted, is to attempt a random solution from the N^V solution space at a time (instead of consecutively, like when using experiments), and hope that it is lucky enough to find a satisfiable one. Given that experiments and PMC cannot be used and

the property to be model checked is qualitative, is the reason why `SINGLE_QUALITATIVE` has to be the `SolvingMode` used by default.

Therefore, as this implementation does not really take advantage of the quantitative capabilities of PRISM nor its features to the fullest, the numeric structures used and the translation considered also do not mesh well with it, especially compared with SMT, currently there is no reason to use PRISM over SMT solving for probabilistic scenarios, since SMT fits with the typed linear algebra approach much better.

After finding a solution, Kodkod is capable of extracting the information from the quantitative solver, including the satisfiability outcome – SAT, UNSAT or UNKNOWN (if a SMT Solver is used) – and, in case the problem is deemed satisfiable, the data used to create an adequate Kodkod instance, also taking into account the quantitative information. Furthermore, after finding an instance, the Kodkod Evaluator was properly adapted to be able to calculate the value of a (quantitative) expression with respect to the quantitative instance, through the assembly process of a constant Numeric Circuit.

Finally, the quantitative extension of Alloy and the Alloy Analyzer was accomplished by conforming to Quantitative Kodkod:

- The Alloy language was also adapted to provide the user with ways to specify quantitative constraints, including giving access to the extended abstract syntax of Kodkod.
- The correspondence between a quantitative Alloy specification and a suitable quantitative Kodkod problem was fully established.
- Integer scope was adapted to, rather than representing the integer bit width (as this is no longer needed) also provide the modeler with another tool, now specifying the maximum weight that any tuple of any instance is allowed to assume when checking a specific command. This is useful to reduce the size of the solution space, if needed.
- Alloy's engine is able to interpret the solutions provided by Kodkod, store its outcomes and represent the instances found using adequately adapted Alloy structures to include the quantitative information.
- During the solving process, the tool is able to report to the user the context-specific details and statistics associated with the problem and the quantitative solvers, analogously to its Boolean counterpart.
- The Analyzer's Evaluator is immediately adapted to the quantitative context, by integrating the quantitative extension of the Kodkod Evaluator.
- As shown in Figures 6.2, 6.3 and 6.4, the Text, Table and Viz representations of Alloy instances within the Alloy Visualizer were successfully adapted to portray the quantitative details. However, the

adaptation of the table representation is not foolproof in the current implementation, not being able to properly display the information for some specific kinds of instance.

Having put the extension developed to the test by applying it to the case studies considered for this project (see Chapter 7), one was able to come to the conclusion that the extension was successfully able to meet all the objectives associated with each case, as the tool is now capable of extracting measurable data implicitly from new and existing Alloy models alike, quantitative constraints can be explicitly imposed, probabilistic analysis can be performed and even the use of probabilistic contract concepts can be taken advantage of.

Even though there is room for this implementation to grow, the current version is already able to perform the solving in reasonable amounts of time, yet slower than SAT solvers when used over the same qualitative problems (as expected). Nevertheless, it performs well in quantitative settings, e.g. when compared with the alternative use of SAT solvers combined with the *multirelations* approach (studied in Section 3.2.1).

Altogether, this project has made a contribution to put the *scalable modeling lemma* “keep definition, change category” (Oliveira and Miraldo, 2016) into practice by enabling shifting over $Mat_{\mathbb{N}_0}$ and LS , performing quantitative analysis on both domains and ultimately ending up with a working Quantitative Alloy, usable on various kinds of problems that could not be analysed before.

8.2 PROSPECT FOR FUTURE WORK

The work described in this dissertation raised a number of research issues that should be addressed in future research polishing and improving the proposed quantitative extension and making it reach new heights.

In Section 7.1, the encoding of Z sparked interest over a kind of matrix different from those considered throughout the dissertation: matrices of non-negative real numbers $Mat_{\mathbb{R}_0^+}$. Given the success of such a setting to still encode the *Bibliometrics* case study properly, while also being able to represent Z as intended under real numbered values, it motivates that $Mat_{\mathbb{R}_0^+}$ should be studied more in-depth. Moreover, perhaps \mathbb{R}_0^+ could be added as a new quantitative analysis domain for Alloy, whose models may reasonably be handled by SMT Solvers using the theory of real numbers. Furthermore, it could be interesting to explore which other kinds of matrix could potentially be used to strengthen Alloy’s power, as well as which solvers might be able to deal with values of that nature.

As seen in Section 2.2, some *Rel* operations could not be immediately shifted into *Mat* in a definitive way. Table 2.2 displays the definitions adopted for some of these operations. Further semantics for these constructs should be explored. One is, for instance, the probabilistic interpretation of $R \cap S$ (“intersection of two left-stochastic matrices”) or, on closer inspection, the meaning of “the intersection of two probabilities for the same event”.¹

¹ As there are multiple ways of relating probabilities in order to make sure which one is more useful or better suited, this needs more research.

In a similar vein, Section 5.5 describes the semantics of the Hadamard division between numeric matrices together with the way division by zero is handled taking into account the numeric matrix representation according to the bounds provided. Currently, division by zero is undefined, causing a potential instance to be considered UNSAT. Consequently, when checking for the validity of an assertion, solutions in which division by zero occurs do not register as valid counterexamples either. Alternatively, instead of deeming division by zero unsatisfiable, $\frac{x}{0} = \infty$ could be regarded, in practice, as the maximum integer/real value. The impact of this decision on instances and counterexamples needs to be evaluated.

As concluded above, the implementation decisions taken in Chapter 5 were deemed suitable to achieve the quantitative extension, as the positive results obtained in several case studies show. Nevertheless, one can only be certain of this once more case studies are carried out, possibly revealing shortcomings. Besides, alternative ways of implementing the same component should be benchmarked, so as to choose the “best” one in terms of problem size, response time, and so on.

Different interpretations for the same construct that arise from the shift into a more expressive domain could also be integrated in parallel, in case they prove to be optimization points of the quantitative extension. For example, the current implementation conventions that Alloy *signatures* keep being represented by Boolean matrices, due to the impact of lifting them to numeric matrices instead, already thoroughly discussed in Chapter 5.

Nonetheless, perhaps by adding a *signature* described by a numeric matrix as new kind of `sig`, alongside the currently implemented (maybe through a new keyword), could improve this extension on different levels. Vice versa, the same line of reasoning could be done with *fields*, instead of having to explicitly constraint $R = \text{drop } R$ to work with R under $\text{Mat}_{\mathbb{B}}$ (as exemplified in Section 6.5), giving the modeler the option to declare an Alloy *field* as Boolean explicitly.

Note that such additions do not increase the expressive power of Alloy, as both unary relations with tuple weights outside the Boolean range and relations characterized by Boolean matrices under a quantitative setting *can* be represented. Although it would, at the very least, save the user from the extra steps needed to specify such relations, promoting the framework’s usability, providing a seamless transition for the regular Alloy user, and the clarity of quantitative specifications. Potentially, it would also allow for a more efficient generation of the constraints associated to these constructs, reducing the load on the solver and improving the performance of the tool overall.

While the current implementation is performing well enough to be used in practice (see Section 7.5), the performance of SMT Solvers and PRISM still do not come nowhere near that of the SAT solvers. Moreover, this straightforward implementation of the assembly of Numeric Circuits, unlike its Boolean counterpart, as mentioned before, does not perform problem optimizations by *symmetry breaking* and *skolemizing* during quantitative analysis. Therefore, it would be of great interest to research how these could be extended to the quantitative realm. This is especially important since, in general, due to the wide range of values that are

now being considered for each primary variable during analysis, it is already harder to reach a conclusion, in contrast to the binary possible value of each literal within a SAT solving setting.

On a similar note, the current concept of Numeric Circuit does not correspond to a full extension of Compact Boolean Circuit. Thus, it would be useful to explore how to achieve compaction of the Numeric Circuit, so that it is also able to detect common components of the circuit and further optimize the problem at hand.

To fully complete the quantitative shift in the Kodkod syntax, a proper quantitative semantics for (reflexive) transitive closure needs to be found. Moreover, the matrix representation of numeric expressions also needs to be adjusted so that, besides being able to explicitly represent any value from the (infinite) domains over which quantitative analysis is performed as is currently implemented, more than one value can be represented at a time within the same numeric matrix.

Concerning SMT Solvers, one could attempt to design potential new theories and/or logical fragments that better accommodate to the typed linear algebra reasoning, Numeric Circuits, or even to the Kodkod abstract syntax itself, in order to reduce response times when handling problems under both $Mat_{\mathbb{N}_0}$ and LS . Moreover, the tool's engine can be enriched by integrating further SMT Solvers that abide to the SMT-LIB standard, through the `SMTSoLver` interface (presented in Section 6.1).

When it comes to PRISM, there is a lot of room for improvement of the current implementation:

- Find ways to reduce the solution space significantly, to make features like experiments usable in practice.
- Explore and fine-tune the solving options to optimize the model checking process taking into account the structure of the model that is being generated automatically.
- Investigate new numeric structures that would better suit the kind of models that PRISM deals with and further adapt the translation into an adequate PRISM model accordingly.

Above all, ensure that the resulting PRISM model abides to the requirements necessary to be subject to useful quantitative solving features offered by PRISM (e.g. parametric model checking).

- Since PRISM struggles with the model finding component, maybe combining the usage of PRISM together with an SMT Solver could bring great results.

By taking advantage of an SMT Solver to determine an instance or reduce the solution space (e.g. outline the intervals of acceptable values that satisfy the model's constraints), PRISM could then be used to verify more complex probabilistic properties with respect to such instance, namely, properties that could not otherwise be verified using SMT solving only.

- Electrum (Macedo et al., 2016) is an Alloy extension to support the verification of temporal properties (LTL/CTL) over Alloy models. If the performance problems associated with PRISM can be resolved,

perhaps the latter could be used in a similar fashion to handle the probabilistic version of temporal properties (PCTL*) in Alloy, whose logic is already innately supported within PRISM.

For clarity, instead of keeping the same nomenclature pattern **Int** (e.g. `IntConstant`, `IntExpression`, ...) for every numeric value/expression/... which is generalized in practice, being evaluated under either integers, natural numbers or real numbers (probabilities) depending on the analysis context, there is the need to clean up the abstract syntax to properly delimit the different kinds of numeric values at both Kodkod and Alloy level. More, extend both the Alloy and Kodkod AST to explicitly be able to specify probabilities, instead of having to be defined by the division between two whole real numbers as in the current implementation.

With the increased expressiveness provided by working over quantitative domains, besides the new operators added to the Alloy language, further useful ones should be explored and implemented in Alloy (and consequently, in Kodkod if necessary) to increase both the specification flexibility and the verification power of the tool.

The Khatri-Rao product, as discussed in Section 7.3, could not be properly implemented as intended, due to the kind of structures used within Kodkod. Managing this product in the desired manner is challenging.

In a similar vein, now that Alloy possesses enough expressiveness to handle probabilistic contracts, one should add support to the measurement calculus of any probabilistic contract in general, rather than the modeler having to explicitly define it at specification-level for every different contract.

Finally, to improve the analysis process and the interaction with the Alloy Analyzer, its GUI can be adjusted and extended, adding quality of life features:

- Fully extend all the visualization types within the Alloy Visualizer to the quantitative context and explore new and alternative ways of better presenting the quantitative information to the user.
- Provide the user more control over the SMT Solvers and PRISM, by adding other configurations to the customizable solving preferences, for example, being able to choose between the different computation methods and model checking algorithms provided by PRISM, managing the quantitative solver's memory usage and so on.
- Implement better ways of manipulating a relation's weights or even adding syntactic sugar to the specification of known integer/probabilistic values in the Alloy specification.
- Research into *scenario exploration* (Macedo et al., 2015) operations for quantitative environments. For instance, during instance enumeration, add the option to skip those which only change in weights, that is, force the solver to ignore solutions which coincide when perceived from the Boolean point-of-view.
- After solving a model under $Mat_{\mathbb{N}_0}$ with no integer scope specified, detect which tuples can have their weight freely varying over an infinite subset of \mathbb{N}_0 in the upcoming instances, and inform the

user of those tuples. If desired, allow the modeler to also fix the value of the weights associated with them, to prevent potential uninteresting instances, where only those primary variables are changing indefinitely.

- When an UNKNOWN response is produced by the SMT Solver, give the user the option of taking a shot in the dark by fixing the weight of some tuple(s), to help the solver arrive at a (UN)SAT response, with respect to the fixed values.

Hopefully, after going over these objectives in the future, one will end up with Quantitative Alloy as a rather sophisticated and powerful *quantitative formal method*.

BIBLIOGRAPHY

- Israa AlAttili, Fred Houben, Georgeta Igna, Steffen Michels, Feng Zhu, and Frits W. Vaandrager. Adaptive scheduling of data paths using uppaal tiga. In [Andova et al. \(2009\)](#), pages 1–11. doi: 10.4204/EPTCS.13.1. URL <https://doi.org/10.4204/EPTCS.13.1>.
- Suzana Andova, Annabelle McIver, Pedro R. D’Argenio, Pieter J. L. Cuijpers, Jasen Markovski, Carroll Morgan, and Manuel Núñez, editors. *Proceedings First Workshop on Quantitative Formal Methods: Theory and Applications, QFM 2009, Eindhoven, The Netherlands, 3rd November 2009*, volume 13 of *EPTCS*, 2009. doi: 10.4204/EPTCS.13. URL <https://doi.org/10.4204/EPTCS.13>.
- Zoe Andrews. Towards a stochastic event-b for designing dependable systems. 07 2009.
- Marco Baroni and Roberto Zamparelli. Nouns are vectors, adjectives are matrices: Representing adjective-noun constructions in semantic space. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1183–1193, Cambridge, MA, October 2010. Association for Computational Linguistics. URL <https://www.aclweb.org/anthology/D10-1115>.
- Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi’c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV ’11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. URL <http://www.cs.stanford.edu/~barrett/pubs/BCD+11.pdf>. Snowbird, Utah.
- Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- Marco Bernardo. Markovian testing equivalence and exponentially timed internal actions. In [Andova et al. \(2009\)](#), pages 13–25. doi: 10.4204/EPTCS.13.2. URL <https://doi.org/10.4204/EPTCS.13.2>.
- Richard S. Bird and Oege de Moor. *Algebra of programming*. Prentice Hall International series in computer science. Prentice Hall, 1997. ISBN 978-0-13-507245-5.
- Stephen Cook and David Mitchell. Finding hard instances of the satisfiability problem: A survey. 35, 01 2000.

- Leonardo de Moura, editor. *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, 2017. Springer. ISBN 978-3-319-63045-8. doi: 10.1007/978-3-319-63046-5. URL <https://doi.org/10.1007/978-3-319-63046-5>.
- Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.
- Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 737–744, Cham, 2014. Springer International Publishing. ISBN 978-3-319-08867-9.
- Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-24605-3.
- M. J. Frade. Propositional logic & sat solvers, February 2019a. HASLab - INESC TEC, Departamento de Informática, Universidade do Minho.
- M. J. Frade. First-order logic & theories, February 2019b. HASLab - INESC TEC, Departamento de Informática, Universidade do Minho.
- M. J. Frade. Smt solvers, February 2019c. HASLab - INESC TEC, Departamento de Informática, Universidade do Minho.
- Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007. doi: 10.1007/978-3-540-73368-3_52. URL https://doi.org/10.1007/978-3-540-73368-3_52.
- Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Annabelle McIver, and Federico Olmedo. Conditioning in probabilistic programming. *CoRR*, abs/1504.00198, 2015. URL <http://arxiv.org/abs/1504.00198>.
- Jun Gu, Paul Purdom, John Franco, and Benjamin Wah. *Algorithms for Satisfiability (SAT) problem: a survey*, volume 35, pages 19–152. 12 1997. doi: 10.1090/dimacs/035/02.
- D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge Mass., 2012. Revised edition, ISBN 0-262-01715-2.

- Susmit Jha, Rhishikesh Limaye, and Sanjit A. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 668–674, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02658-4.
- Yousef Kilani, Mohammad Bsoul, Ayoub Alsarhan, and Ahmad Al-Khasawneh. A survey of the satisfiability-problems solving algorithms. *International Journal of Advanced Intelligence Paradigms*, 5:233–256, 09 2013. doi: 10.1504/IJAIP.2013.056447.
- M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
- Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. ISSN 1574-0617. doi: 10.3233/SAT190075. URL <https://doi.org/10.3233/SAT190075>. 2-3.
- Yongming li. Quantitative model checking of linear-time properties based on generalized possibility measures. *Fuzzy Sets and Systems*, 320:17–39, 08 2017. doi: 10.1016/j.fss.2017.03.0120165.
- Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. Formal verification of quantum algorithms using quantum hoare logic. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 187–207, Cham, 2019. Springer International Publishing. ISBN 978-3-030-25543-5.
- Nuno Macedo, Alcino Cunha, and Tiago Guimarães. Exploring scenario exploration. In *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering*. Springer, Springer, 2015.
- Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, page 373–383, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. doi: 10.1145/2950290.2950318. URL <https://doi.org/10.1145/2950290.2950318>.
- S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.
- Paulo Mateus and Amilcar Sernadas. Reasoning about quantum systems. volume 3229, pages 239–251, 09 2004. doi: 10.1007/978-3-540-30227-8_22.
- Baolu Meng, Andrew Reynolds, Cesare Tinelli, and Clark W. Barrett. Relational constraint solving in SMT. In [de Moura \(2017\)](#), pages 148–165. ISBN 978-3-319-63045-8. doi: 10.1007/978-3-319-63046-5_10. URL https://doi.org/10.1007/978-3-319-63046-5_10.

- Aleksandar Milicevic and Daniel Jackson. Preventing arithmetic overflows in alloy. In John Derrick, John Fitzgerald, Stefania Gnesi, Sarfraz Khurshid, Michael Leuschel, Steve Reeves, and Elvinia Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and Z*, pages 108–121, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-30885-7.
- David Monniaux. A survey of satisfiability modulo theory. 06 2016.
- Ukachukwu Ndukwu. Quantitative safety: Linking proof-based verification with model checking for probabilistic systems. In Andova et al. (2009), pages 27–39. doi: 10.4204/EPTCS.13.3. URL <https://doi.org/10.4204/EPTCS.13.3>.
- Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1): 53–58, 2014. doi: 10.3233/sat190101. URL <https://doi.org/10.3233/sat190101>.
- Libero Nigro and Paolo Sciammarella. Qualitative and quantitative model checking of distributed probabilistic timed actors. *Simulation Modelling Practice and Theory*, 87, 07 2018. doi: 10.1016/j.simpat.2018.07.011.
- J.N. Oliveira. Towards a linear algebra of programming. *FAoC*, 24(4-6):433–458, 2012a.
- J.N. Oliveira. Measuring probabilistic contracts, Feb 2017a. Presented at [IFIP WG 2.1 #75 Meeting](#), Montevideo, 20-24 Feb (slides available from the 2.1 website.).
- J.N. Oliveira and H. Macedo. The data cube as a typed linear algebra operator. In *Proc. of the 16th Int. Symposium on Database Programming Languages*, DBPL '17, page 6:1–6:11, New York, NY, USA, 2017. ACM, ACM. ISBN 978-1-4503-5354-0. doi: 10.1145/3122831.3122834. URL <http://doi.acm.org/10.1145/3122831.3122834>. n/a.
- José N. Oliveira. Typed linear algebra for weighed (probabilistic) automata. In Nelma Moreira and Rogério Reis, editors, *Implementation and Application of Automata*, pages 52–65, Berlin, Heidelberg, 2012b. Springer Berlin Heidelberg. ISBN 978-3-642-31606-7.
- José Nuno Oliveira and Victor Cacciari Miraldo. "keep definition, change category" - A practical approach to state-based system calculi. *J. Log. Algebr. Meth. Program.*, 85(4):449–474, 2016. doi: 10.1016/j.jlamp.2015.11.007. URL <https://doi.org/10.1016/j.jlamp.2015.11.007>.
- José N. Oliveira. Going quantitative in software modeling. TRUST 2nd Workshop, 2017b.
- José N. Oliveira. *Program Design by Calculation*. 2019. Draft of textbook in preparation, current version: Oct. 2019. Informatics Department, University of Minho.
- Dave Parker. Probabilistic model checking, 2011. Department of Computer Science, University of Oxford.

- solid IT. Db-engines ranking, 2020. URL <https://db-engines.com/en/ranking/>. [Online; accessed 29-January-2020].
- Peiyuan Sun, Zinovy Diskin, Michał Antkiewicz, and Krzysztof Czarnecki. Modeling and reasoning with multirelations, and their encoding in alloy. In *16th International Workshop in OCL and Textual Modeling*, 10/2016 2016. URL http://oclworkshop.github.io/2016/papers/OCL16_paper_10.pdf.
- Alfred Tarski and Steven Givant. *A Formalization of Set Theory without Variables*. American Mathematical Society, December 1987. doi: 10.1090/coll/041. URL <https://doi.org/10.1090/coll/041>.
- Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors. *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, 2019. Springer. ISBN 978-3-030-30941-1. doi: 10.1007/978-3-030-30942-8. URL <https://doi.org/10.1007/978-3-030-30942-8>.
- Emina Torlak and Daniel Jackson. The design of a relational engine emina torlak and. 2006.
- Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-71209-1.
- Hao Wang and Wendy MacCaull. Verifying real-time systems using explicit-time description methods. In [Andova et al. \(2009\)](#), pages 67–78. doi: 10.4204/EPTCS.13.6. URL <https://doi.org/10.4204/EPTCS.13.6>.
- Wikipedia contributors. Bayesian network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Bayesian_network&oldid=926429150, 2019. [Online; accessed 19-December-2019].

A

LISTINGS

A.1 BIBLIOMETRICS

A.1.1 *Alloy Model*

```
sig Keyword, Medium{}
```

```
sig Author{ Q : set Keyword }
```

```
sig Paper{  
  C : set Paper,  
  K : set Keyword,  
  A : set Author,  
  m : one Medium,  
  S : set Keyword  
}
```

```
fact{  
  no C & iden  
  
  S = K & ~C.K  
  Q = ~A.S  
}
```

A.1.2 *Quantitative Alloy Model*

```
sig Keyword, Medium{}
```



```

sig Author{
  Q : set Keyword,
  Z : set Keyword
}

sig Paper{
  C : set Paper,
  K : set Keyword,
  A : set Author,
  m : one Medium,
  S : set Keyword
}

fact{
  no C & iden

  drop S = K & ~C.K
  all p : Paper, k : Keyword |
    (p -> k) in S implies #(p <: S.k) = #(p <: (~C.K).k)

  Q = ~A.S

  Z = Q fun/div (Author -> Paper).S

  drop K = K
  drop A = A
  drop C = C
}

```

A.1.3 *Quantitative Instance*

```

---INSTANCE---
integers={1}
univ={1, Author$0, Author$1, Author$2, Keyword$0, Keyword$1, Keyword$2,
Keyword$3, Keyword$4, Keyword$5, Medium$0, Medium$1, Paper$0, Paper$1,

```

```

Paper$2, Paper$3, Paper$4, Paper$5, Paper$6, Paper$7, Paper$8, Paper$9}
Int={1}
seq/Int={}
String={}
none={}
this/Keyword={Keyword$0, Keyword$1, Keyword$2, Keyword$3, Keyword$4,
Keyword$5}
this/Medium={Medium$0, Medium$1}
this/Author={Author$0, Author$1, Author$2}
this/Author<:Q={({Author$0->Keyword$0,3), (Author$0->Keyword$1,1),
(Author$0->Keyword$2,2), (Author$0->Keyword$4,1), (Author$1->Keyword$0,6),
(Author$1->Keyword$1,6), (Author$1->Keyword$2,4), (Author$1->Keyword$4,16),
(Author$1->Keyword$5,2), (Author$2->Keyword$0,6), (Author$2->Keyword$1,3),
(Author$2->Keyword$2,2), (Author$2->Keyword$4,1)})
this/Paper={Paper$0, Paper$1, Paper$2, Paper$3, Paper$4, Paper$5, Paper$6,
Paper$7, Paper$8, Paper$9}
this/Paper<:C={
Paper$0->Paper$3, Paper$0->Paper$5, Paper$0->Paper$8, Paper$1->Paper$0,
Paper$1->Paper$2, Paper$1->Paper$6, Paper$1->Paper$8, Paper$2->Paper$0,
Paper$2->Paper$8, Paper$3->Paper$0, Paper$3->Paper$7, Paper$4->Paper$2,
Paper$4->Paper$3, Paper$4->Paper$5, Paper$4->Paper$7, Paper$5->Paper$7,
Paper$6->Paper$0, Paper$6->Paper$1, Paper$6->Paper$2, Paper$6->Paper$3,
Paper$6->Paper$5, Paper$6->Paper$7, Paper$6->Paper$9, Paper$7->Paper$4,
Paper$7->Paper$5, Paper$7->Paper$8, Paper$8->Paper$0, Paper$8->Paper$2,
Paper$8->Paper$3, Paper$8->Paper$5, Paper$8->Paper$6, Paper$8->Paper$7,
Paper$9->Paper$2, Paper$9->Paper$7, Paper$9->Paper$8}
this/Paper<:K={Paper$0->Keyword$0, Paper$0->Keyword$1, Paper$0->Keyword$2,
Paper$0->Keyword$4, Paper$0->Keyword$5, Paper$1->Keyword$0,
Paper$1->Keyword$3, Paper$1->Keyword$4, Paper$2->Keyword$1,
Paper$2->Keyword$2, Paper$2->Keyword$4, Paper$3->Keyword$0,
Paper$3->Keyword$1, Paper$3->Keyword$2, Paper$4->Keyword$0,
Paper$4->Keyword$4, Paper$4->Keyword$5, Paper$5->Keyword$0,
Paper$5->Keyword$1, Paper$6->Keyword$2, Paper$6->Keyword$4,
Paper$7->Keyword$1, Paper$8->Keyword$0, Paper$8->Keyword$4,
Paper$8->Keyword$5, Paper$9->Keyword$0, Paper$9->Keyword$1,
Paper$9->Keyword$4}

```

```

this/Paper<:A={Paper$0->Author$1, Paper$1->Author$1, Paper$1->Author$2,
Paper$2->Author$1, Paper$3->Author$0, Paper$3->Author$2, Paper$4->Author$0,
Paper$4->Author$1, Paper$5->Author$2, Paper$6->Author$1, Paper$7->Author$1,
Paper$8->Author$1, Paper$9->Author$0}
this/Paper<:m={Paper$0->Medium$1, Paper$1->Medium$0, Paper$2->Medium$0,
Paper$3->Medium$1, Paper$4->Medium$1, Paper$5->Medium$0, Paper$6->Medium$0,
Paper$7->Medium$0, Paper$8->Medium$1, Paper$9->Medium$0}
this/Paper<:S={(Paper$0->Keyword$0,3), (Paper$0->Keyword$1,2),
(Paper$0->Keyword$2,3), (Paper$0->Keyword$4,4), (Paper$0->Keyword$5,1),
(Paper$1->Keyword$4,1), (Paper$2->Keyword$1,1), (Paper$2->Keyword$2,1),
(Paper$2->Keyword$4,5), (Paper$3->Keyword$0,3), (Paper$3->Keyword$1,1),
(Paper$3->Keyword$2,2), (Paper$5->Keyword$0,3), (Paper$5->Keyword$1,2),
(Paper$6->Keyword$4,2), (Paper$7->Keyword$1,3), (Paper$8->Keyword$0,3),
(Paper$8->Keyword$4,4), (Paper$8->Keyword$5,1), (Paper$9->Keyword$4,1)}

```

A.2 FOOTBALL CHAMPIONSHIP

A.2.1 Alloy Model

```
sig Team{}
```

```
sig Date{}
```

```

sig Game{
    home : one Team,
    away : one Team,
    date : one Date,
    s : one Game
}

```

```

fact{
    --(a) No team can play two games on the same date;
    (away + home) . ~(away + home) & date . ~date in iden

    --(b) All teams play against each other but not against themselves
    ~home . away = (Team -> Team) - iden
}

```

```

--(c) For each home game there is another game away
-- involving the same two teams
away . ~away & home . ~home in iden

-- Game isomorphism
s = away . ~home & home . ~away
}

```

A.2.2 SMT2-LIB Specification

```

(set-logic QF_UFLIA)
...
(declare-const g111 Int)
(declare-const g112 Int)
...
; Non-negativity constraints
(assert (>= g111 0))
(assert (>= g112 0))
...
; No team can play two games on the same date
; Team 1, Date 1
(assert (<= (+ g111 g121 g131 g141 g211 g311 g411) 1))
...
; All teams play against each other but not against themselves
; Team 1
(assert (= (+ g111 g112 g113 g114 g115 g116) 0))
; Game: Team 1 (home) vs Team 2 (away)
(assert (>= (+ g121 g122 g123 g124 g125 g126) 1))
...
; For each home game there is another game away
; involving the same two teams
; Team 1 and Team 2
(assert
  (=

```

```

        (+ g121 g122 g123 g124 g125 g126)
        (+ g211 g212 g213 g214 g215 g216)
    )
)
...
(declare-const w1 Int)
(declare-const w2 Int)
...
(declare-const l1 Int)
(declare-const l2 Int)
...
(declare-const w12 Int)
(declare-const w13 Int)
...
(declare-const l12 Int)
(declare-const l13 Int)
...
; Non-negativity constraints
(assert (>= w12 0))
(assert (>= w13 0))
...
; Team 1
(assert
  (=
    (+ w1 l1)
    (+
      g111 g112 g113 g114 g115 g116
      g121 g122 g123 g124 g125 g126
      g131 g132 g133 g134 g135 g136
      g141 g142 g143 g144 g145 g146
      g211 g212 g213 g214 g215 g216
      g311 g312 g313 g314 g315 g316
      g411 g412 g413 g414 g415 g416
    )
  )
)
)

```

```

...
(assert (= w1 (+ w12 w13 w14)))
...
(assert (= l1 (+ l12 l13 l14)))
...
(assert (= w12 l21))
(assert (= w13 l31))
(assert (= w14 l41))
...
(assert
  (=
    (+ w12 l12)
    (+
      g121 g122 g123 g124 g125 g126
      g211 g212 g213 g214 g215 g216
    )
  )
)
)
...
(check-sat)
(get-model)
...

```

A.2.3 Alloy Model using Quantitative Invariants

```

sig Team{}

sig Date{}

sig Game{
  home : one Team,
  away : one Team,
  date : one Date,
  s : one Game
}

```

```

fact{
  --(a) No team can play two games on the same date;
  ~(away + home) . date <= Team -> Date

  --(b) All teams play against each other but not against themselves
  ~home . away = (Team -> Team) - iden

  --(c) For each home game there is another game away
  -- involving the same two teams
  away . ~away & home . ~home in iden

  -- Game isomorphism
  s = away . ~home & home . ~away
}

assert oneGameData{ (away + home) . ~(away + home) & date . ~date in iden }
check oneGameData
for exactly
  2 Team,
  2 Game,
  2 Date
check oneGameData
for exactly
  3 Team,
  6 Game,
  6 Date
check oneGameData
for exactly
  4 Team,
  12 Game,
  6 Date

```

A.2.4 *Quantitative Alloy Model with Quantitative Relations*

```
sig Team{ History : set Result }
```

```
sig Date{}
```

```
sig Game{
  home : one Team,
  away : one Team,
  date : one Date,
  s : one Game
}
```

```
abstract sig Result{}
```

```
one sig Win, Lose extends Result{}
```

```
fact{
```

```
--(a) No team can play two games on the same date;
```

```
~(away + home) . date <= Team -> Date
```

```
--(b) All teams play against each other but not against themselves
```

```
~home . away = (Team -> Team) - iden
```

```
--(c) For each home game there is another game away
```

```
-- involving the same two teams
```

```
away . ~away & home . ~home in iden
```

```
-- Game isomorphism
```

```
s = away . ~home & home . ~away
```

```
-- There is a winner for every loser
```

```
#History.Win = #History.Lose
```

```
-- Every team either wins or loses each game in which they take part
```

```
all t : Team | #(home + away).t = #t.History
```

```
}
```


A.2.5 Alloy Model with Multirelations

```

open multi
open mrel[Team, Result] as History

one sig Unit{}

sig Team{}

sig Date{}

sig Game{
  home : one Team,
  away : one Team,
  date : one Date,
  s : one Game
}

abstract sig Result{}
one sig Win, Lose extends Result{}

fact{
  (away + home) . ~(away + home) & date . ~date in iden
  ~home . away = (Team -> Team) - iden
  away . ~away & home . ~home in iden
  s = away . ~home & home . ~away

  #History/get :> Win = #History/get :> Lose
  all t : Team | #(home + away).t = #t.History/get
}

```

A.3 SPRINKLER

A.3.1 *Initial Scenario*

dtmc

module rain

r : [0..2] init 2;

[] r=2 -> 0.8 : (r' = 0) + 0.2 : (r' = 1);

[] r=0 | r=1 -> (r'=r);

endmodule

module sprinkler

s : [0..2] init 2;

[] r=0 & s=2 -> 0.4 : (s' = 1) + 0.6 : (s' = 0);

[] r=1 & s=2 -> 0.01 : (s' = 1) + 0.99 : (s' = 0);

[] s=0 | s=1 -> (s'=s);

endmodule

module grass

g : [0..2] init 2;

[] s=0 & r=0 & g=2 -> (g'=0);

[] s=0 & r=1 & g=2 -> 0.8 : (g' = 1) + 0.2 : (g' = 0);

[] s=1 & r=0 & g=2 -> 0.9 : (g' = 1) + 0.1 : (g' = 0);

[] s=1 & r=1 & g=2 -> 0.99 : (g' = 1) + 0.01 : (g' = 0);

[] g=0 | g=1 -> (g'=g);

endmodule

A.3.2 *Unknown Sprinkler*

dtmc

```

module rain
  r : [0..2] init 2;

  [] r=2 -> 0.8 : (r' = 0) + 0.2 : (r' = 1);
  [] r=0 | r=1 -> (r'=r);
endmodule

const double s10;
const double s01;

module sprinkler
  s : [0..2] init 2;

  [] r=0 & s=2 -> s10 : (s' = 1) + (1-s10) : (s' = 0);
  [] r=1 & s=2 -> (1-s01) : (s' = 1) + s01 : (s' = 0);
  [] s=0 | s=1 -> (s'=s);
endmodule

module grass
  g : [0..2] init 2;

  [] s=0 & r=0 & g=2 -> (g'=0);
  [] s=0 & r=1 & g=2 -> 0.8 : (g' = 1) + 0.2 : (g' = 0);
  [] s=1 & r=0 & g=2 -> 0.9 : (g' = 1) + 0.1 : (g' = 0);
  [] s=1 & r=1 & g=2 -> 0.99 : (g' = 1) + 0.01 : (g' = 0);
  [] g=0 | g=1 -> (g'=g);
endmodule

```

A.3.3 Alloy Model of a Bayesian Network

```
one sig Unit{ rain : one R }
```

```
abstract sig R{
  sprinkler : one S,
```

```

    grass : S set -> one G
}

one sig Rain, NoRain extends R{}

abstract sig S, G{}
one sig On, Off extends S{}
one sig Wet, Dry extends G{}

fact{
  //rain
  #(Unit.rain :=> Rain) = div[2, 10]

  //sprinkler
  #(NoRain.sprinkler :=> On) = div[4, 10]
  #(Rain.sprinkler :=> On) = div[1, 100]

  //grass
  one Off.(NoRain.grass) :=> Dry
  #(Off.(Rain.grass) :=> Dry) = div[2, 10]
  #(On.(NoRain.grass) :=> Dry) = div[1, 10]
  #(On.(Rain.grass) :=> Dry) = div[1, 100]
}

```

A.3.4 Bayes' Theorem in Alloy

```

one sig Unit{ rain : one R }

abstract sig R{
  sprinkler : one S,
  spID : one RS
}

one sig Rain, NoRain extends R{}

```

```

abstract sig S, G{}
one sig On, Off extends S{}
one sig Wet, Dry extends G{}

// S x R
abstract sig RS{
  grass : one G
}
one sig RainOff, RainOn, NoRainOff, NoRainOn extends RS{}

fact{
  // rain
  #(Unit.rain :=> Rain) = div[2, 10]

  // grass
  one NoRainOff.grass :=> Dry
  #(RainOff.grass :=> Dry) = div[2, 10]
  #(NoRainOn.grass :=> Dry) = div[1, 10]
  #(RainOn.grass :=> Dry) = div[1, 100]

  // Khatri-Rao product
  #(Rain.spID :=> RainOn) = #(Rain.sprinkler :=> On)
  #(Rain.spID :=> RainOff) = #(Rain.sprinkler :=> Off)
  #(NoRain.spID :=> NoRainOn) = #(NoRain.sprinkler :=> On)
  #(NoRain.spID :=> NoRainOff) = #(NoRain.sprinkler :=> Off)
}

fun grassWet : G {
  Unit.rain.spID.grass :=> Wet
}

// Forwards reasoning: P(g = 1 | r = 1)
fun gwIfRain : G {
  Rain.spID.grass :=> Wet
}

```

```

// Backwards reasoning through Bayes' theorem
//  $P(r = 1 \mid g = 1) = P(g = 1 \mid r = 1) * P(r = 1) / P(g = 1)$ 
fun rainIfGW : Int {
    mul[#gwIfRain, div[#(Unit.rain :=> Rain), #grassWet]]
}

run{
    // sprinkler
    #(NoRain.sprinkler :=> On) = div[4, 10]
    #(Rain.sprinkler :=> On) = div[1, 100]
}

```

A.3.5 Probabilistic Contract

```

one sig Unit{
    rain : one R,
    delta : one R
}

abstract sig R{
    sprinkler : one S,
    spID : one RS,
    f : one SG
}

one sig Rain, NoRain extends R{}

abstract sig S, G{}
one sig On, Off extends S{}
one sig Wet, Dry extends G{}

abstract sig RS{
    grass : one G
}

one sig RainOff, RainOn, NoRainOff, NoRainOn extends RS{}

```

```

abstract sig SG{}
one sig OffDry, OffWet, OnDry, OnWet extends SG{}

fact{
  #(NoRain.sprinkler :> On) = div[4, 10]
  #(Rain.sprinkler :> On) = div[1, 100]

  one NoRainOff.grass :> Dry
  #(RainOff.grass :> Dry) = div[2, 10]
  #(NoRainOn.grass :> Dry) = div[1, 10]
  #(RainOn.grass :> Dry) = div[1, 100]

  #(Rain.spID :> RainOn) = #(Rain.sprinkler :> On)
  #(Rain.spID :> RainOff) = #(Rain.sprinkler :> Off)
  #(NoRain.spID :> NoRainOn) = #(NoRain.sprinkler :> On)
  #(NoRain.spID :> NoRainOff) = #(NoRain.sprinkler :> Off)

  let gis = spID.grass {
    #(NoRain.f :> OffDry)
      = mul[#(NoRain.sprinkler :> Off), #(NoRain.gis :> Dry)]
    #(NoRain.f :> OffWet)
      = mul[#(NoRain.sprinkler :> Off), #(NoRain.gis :> Wet)]
    #(NoRain.f :> OnDry)
      = mul[#(NoRain.sprinkler :> On), #(NoRain.gis :> Dry)]
    #(NoRain.f :> OnWet)
      = mul[#(NoRain.sprinkler :> On), #(NoRain.gis :> Wet)]
    #(Rain.f :> OffDry)
      = mul[#(Rain.sprinkler :> Off), #(Rain.gis :> Dry)]
    #(Rain.f :> OffWet)
      = mul[#(Rain.sprinkler :> Off), #(Rain.gis :> Wet)]
    #(Rain.f :> OnDry)
      = mul[#(Rain.sprinkler :> On), #(Rain.gis :> Dry)]
    #(Rain.f :> OnWet)
      = mul[#(Rain.sprinkler :> On), #(Rain.gis :> Wet)]
  }
}

```

```

}

fun grass_wet : SG{ OffWet + OnWet }
fun raining : R{ Rain }

fun measureContract : Unit{
  delta.(f.grass_wet fun/mul raining) fun/div delta.raining
}

assert contract{
  #measureContract = div[8019, 10000]
}
check contract

```

A.4 EXAMPLE OF AN ALLOY PROBABILISTIC CONTRACT

```

abstract sig A{ f : one B }
abstract sig B{}

one sig Unit{
  delta : one A
}

one sig a1, a2, a3 extends A{}
one sig b1, b2 extends B{}

fact PF{
  #(a1.f :=> b1) = div[7, 10]
  #(a2.f :=> b1) = div[1, 100]
  one a3.f :=> b1
}

fun measureContract[p' : A, q' : B] : Int {
  let p = drop p', q = drop q' | #delta.(f.q fun/mul p) fun/div #delta.p

```



```

}

run knownDelta{
  #(delta :> a1) = div[1, 10]
  #(delta :> a2) = div[2, 10]
  -- #(delta :> a3) = div[7, 10]
}

assert contract{
  measureContract[a1 + a2, b2] >= 8 fun/div 10
}
check contract

```

A.5 BUNDLING

A.5.1 Alloy Specification using Multirelations

```

open multi

open mrel[Bundle, Item] as Contains
open mrel[Item, Category] as BelongsTo
open mrel[Bundle, Category] as Result

abstract sig Bundle {}
one sig B1, B2 extends Bundle {}
abstract sig Item {
  belongsTo: set Category
}
one sig Bread, Butter, Milk extends Item {}
fact {
  Bread.belongsTo = Bakery
  Butter.belongsTo = Dairy
  Milk.belongsTo = Dairy
}
abstract sig Category {}

```

```
one sig Dairy, Bakery extends Category {}
```

```
fact {
  BelongsTo/liftedFrom[belongsTo]
  Result/composedFrom[Contains/get, BelongsTo/get]
  all b : Bundle | #(b<:Result/get:>Dairy) >= 2
  all b : Bundle | #(b.(drop[Result/get])) >= 2
}
```

```
assert AllHaveBread {
  all b: Bundle | some drop[b<:Contains/get].Bread
}
```

```
run {} for 6 Contains/Head, 3 BelongsTo/Head, 20 Result/Head
check AllHaveBread for 6 Contains/Head, 3 BelongsTo/Head, 20 Result/Head
```

A.5.2 Quantitative Alloy Specification

```
abstract sig Bundle{
  Contains : set Item,
  Result : set Category
}
one sig B1, B2 extends Bundle {}

abstract sig Item{
  BelongsTo : one Category
}
one sig Bread, Butter, Milk extends Item{}

abstract sig Category{}
one sig Dairy, Bakery extends Category {}

fact{
  Result = Contains . BelongsTo
```

```

    Bread.BelongsTo = Bakery
    Butter.BelongsTo = Dairy
    Milk.BelongsTo = Dairy

    all b : Bundle | #(b.Result :> Dairy) >= 2
    all b : Bundle | #drop(b.Result) >= 2
}

assert AllHaveBread {
    all b: Bundle | some drop(b <: Contains).Bread
}
check AllHaveBread for 10 Int
run{} for 10

```

A.5.3 Generalized Alloy Specification using Multirelations

```

open multi

open multirelations/mrel[Bundle, Item] as Contains
open multirelations/mrel[Item, Category] as BelongsTo
open multirelations/mrel[Bundle, Category] as Result

sig Bundle {}
sig Item { belongsTo: set Category }
sig Category {}

fact {
    BelongsTo/liftedFrom[belongsTo]
    Result/composedFrom[Contains/get, BelongsTo/get]
    some c : Category | all b : Bundle | #(b<:Result/get:>c) >= 2
    all b : Bundle | #(b.(drop[Result/get])) >= 2
}

```

A.5.4 *Generalized Quantitative Alloy Specification*

```
sig Bundle{
  Contains : set Item,
  Result : set Category
}
sig Item{ BelongsTo : one Category }
sig Category{}

fact{
  Result = Contains . BelongsTo

  some c : Category | all b : Bundle | #(b <: Result :> c) >= 2
  all b : Bundle | #drop(b.Result) >= 2
}
```

A.6 BENCHMARK

Bibliometrics			
	Paper = 3 Author = 3 Medium = 1 Keyword = 1	Paper = 10 Author = 3 Medium = 2 Keyword = 3 Int = 6	Paper = 15 Author = 6 Medium = 3 Keyword = 5 Int = 6
SAT	20	94	234
	13	55	211
	10	61	256
	5	45	348
	5	35	302
	4	64	175
	9.50	59.00	254.33
	Paper = 3 Author = 3 Medium = 1 Keyword = 1	Paper = 10 Author = 3 Medium = 2 Keyword = 3	Paper = 15 Author = 6 Medium = 3 Keyword = 5
SMT	194	5566	24294
	102	5667	24131
	107	5741	23872
	90	5681	24179
	108	7418	24011
	99	5706	24817
	116.67	5963.17	24217.33
SMT <i>(quantitative adaptation)</i>	191	5264	54832
	100	5283	54873
	91	5425	54846
	105	6083	54336
	100	6652	56751
	97	5587	58101
	114.00	5715.67	55623.17
Football Championship			
	Team = 2 Game ≤ 2 Date ≤ 2	Team = 3 Game ≤ 6 Date ≤ 6	Team = 4 Game ≤ 12 Date ≤ 6
SAT	14	33	146
	6	19	95
	8	14	49
	11	19	71
	3	14	55
	9	17	70
	8.5	19.33	81
SMT	> 7 minutes → unknown	5929	130271
	-	6247	132514
	-	6304	130833
	-	6129	129268
	-	5669	127658
	-	5984	128628
	-	6043.67	129862

Football Championship - Quantitative Invariant			
	Team = 2 Game ≤ 2 Date ≤ 2	Team = 3 Game ≤ 6 Date ≤ 6	Team = 4 Game ≤ 12 Date ≤ 6
SMT	230	3862	57748
	128	3737	54293
	135	3818	59090
	115	3819	53984
	125	4491	55854
	133	3752	56238
	144.33	3913.17	56201.17
Football Championship with History			
	Team = 2 Game ≤ 2 Date ≤ 2	Team = 3 Game ≤ 6 Date ≤ 6	Team = 4 Game ≤ 12 Date ≤ 6
SMT	340	3403	55350
	241	3523	55901
	255	3452	55615
	229	3463	55281
	286	5007	57578
	211	3448	55566
	260.33	3716.00	55881.83
	Team = 2 Game ≤ 2 Date ≤ 2 History/Head ≤ 4 Int = 4	Team = 3 Game ≤ 6 Date ≤ 6 History/Head ≤ 12 Int = 5	Team = 4 Game ≤ 12 Date ≤ 6 History/Head ≤ 24 Int = 6
SAT <i>multirelations</i>	35	129	725
	10	72	605
	8	66	663
	12	71	744
	8	84	653
	8	40	513
	13.50	77.00	650.50

Bayesian Network - Unknown sprinkler			
<i>any sprinkler</i>	P(g = 1) = 50%	P(g = 1) ≥ 90%	P(g = 1) < 20%
61	43	42	39
40	51	79	85
45	40	55	47
45	42	35	35
38	38	40	42
37	38	38	51
44.33	42.00	48.17	49.83
Measure Contract			
check ≥ 80%	run = 78.3%	run > 78.3%	run < 78.3%
33	23	22	21
19	19	31	21
25	30	21	27
19	20	32	23
21	25	25	36
23	29	19	18
23.33	24.33	25.00	24.33

Bundling			
	Bundle = 3 Item = 5 Category = 3 Contains/Head = 10 BelongsTo/Head = 10 Result/Head = 10 Int = 6	Bundle = 5 Item = 10 Category = 3 Contains/Head = 30 BelongsTo/Head = 30 Result/Head = 30 Int = 6	Bundle = 10 Item = 20 Category = 5 Contains/Head = 40 BelongsTo/Head = 40 Result/Head = 50 Int = 8
SAT <i>multirelations</i>	5675	> 35 minutes without response	-
	5235	-	-
	5385	-	-
	5766	-	-
	5541	-	-
	5440	-	-
	5507.00	-	-
	Bundle = 3 Item = 5 Category = 3	Bundle = 5 Item = 10 Category = 3	Bundle = 10 Item = 20 Category = 5
SMT	192	465	5807
	154	605	5827
	120	412	5803
	121	385	5768
	135	450	7535
	113	402	6136
	139.17	453.17	6146.00
	Contains/Head = 6 BelongsTo/Head = 3 Result/Head = 20	check Contains/Head = 6 BelongsTo/Head = 3 Result/Head = 20	
SAT <i>multirelations</i>	125	307	
	122	372	
	156	305	
	94	403	
	112	425	
	89	347	
116.33	359.83		
	for 10	check for 10 Int	
SMT	46	40	
	35	29	
	29	44	
	53	42	
	45	44	
	29	26	
39.50	37.50		

