

Automatic Test Generation for Data-Driven Applications

Michaël MARCOZZI

Namur, le 24 Octobre 2014

Jury de Thèse

Pr. Naji Habra. *Université de Namur, Belgique.* – PRÉSIDENT
Pr. Wim Vanhoof. *Université de Namur, Belgique.* – PROMOTEUR, SECRÉTAIRE
Pr. Jean-Luc Hainaut. *Université de Namur, Belgique.* – CO-PROMOTEUR
Dr. Alexander Serebrenik. *T.U. Eindhoven, Pays-Bas.* – RAPPORTEUR EXTERNE
Pr. Jef Wijsen. *Université de Mons, Belgique.* – RAPPORTEUR EXTERNE
Pr. Vincent Englebert. *Université de Namur, Belgique.* – RAPPORTEUR INTERNE

Doctorat *financé par* une bourse d'Aspirant octroyée par le F.R.S.-FNRS

Thèse *présentée en vue de l'obtention du grade de*
Docteur en Sciences (orientation: Science Informatique)

Automatic Test Generation for Data-Driven Applications

Michaël MARCOZZI

Namur, October 24, 2014

Ph.D. Thesis Committee

Pr. Naji Habra. *University of Namur, Belgium.* – CHAIR
Pr. Wim Vanhoof. *University of Namur, Belgium.* – SUPERVISOR, SECRETARY
Pr. Jean-Luc Hainaut. *University of Namur, Belgium.* – CO-SUPERVISOR
Dr. Alexander Serebrenik. *T.U. Eindhoven, Netherlands.* – EXTERNAL REVIEWER
Pr. Jef Wijsen. *University of Namur, Belgium.* – EXTERNAL REVIEWER
Pr. Vincent Englebert. *University of Namur, Belgium.* – INTERNAL REVIEWER

Ph.D. *funded by* a Fellowship scholarship granted by F.R.S.-FNRS

Thesis *presented for the degree of*
Doctor of Science (speciality: Computer Science)

Résumé. Tous les êtres humains commettent des erreurs. Les programmeurs sont des êtres humains. Donc les programmeurs commettent des erreurs. En apparence anodin, voici un syllogisme dont les conséquences peuvent être catastrophiques pour la société. Alors que des pans de plus en plus larges de notre vie économique et sociale reposent sur l'aide des ordinateurs, les bugs informatiques causés par des erreurs de programmation posent un risque de plus en plus grave. Un exemple qui a marqué les esprits est l'explosion en vol de la fusée Ariane 5, due à un dépassement d'entier dans la mémoire de l'ordinateur de bord.

Dans la pratique actuelle de l'ingénierie du logiciel, la première stratégie mise en oeuvre pour détecter les erreurs de programmation est le test. Avant sa mise en production, le logiciel est ainsi testé dans un ensemble de scénarios réels, afin d'identifier au mieux les éventuels bugs. Néanmoins, dans sa mise en oeuvre concrète, le processus permettant de tester un logiciel est complexe, répétitif et coûteux. C'est pourquoi chercheurs et industriels travaillent depuis longtemps à l'automatiser. L'idée est notamment que l'ordinateur aide le programmeur à sélectionner automatiquement les scénarios de test les plus pertinents. Une approche intéressante pour y arriver est l'exécution symbolique, aujourd'hui utilisée notamment à la Nasa et chez Microsoft. Techniquement, étant donné une unité de code à tester, l'exécution symbolique permet de faire générer par l'ordinateur des valeurs de test pour ses paramètres d'entrée, couvrant un ensemble représentatif de chemins d'exécution dans le code.

Le présent travail propose une extension de l'exécution symbolique, qui rend possible l'utilisation de cette approche dans un nouveau contexte : le test d'unités de code transactionnelles et orientées données. Ce type d'unités de code, fréquent en pratique, se caractérise par une interaction brève et ciblée avec une base de donnée relationnelle, stockant de manière structurée de grandes quantités d'informations. Détecter les bugs dans ce type d'unité de code est crucial, car cela peut éviter la corruption de données potentiellement très importantes. Pour y arriver, notre travail retourne aux fondements théoriques des bases de données relationnelles et propose un mécanisme d'exécution symbolique relationnel pour SQL. Cela permet de générer automatiquement des données d'entrée, y compris du contenu valide pour la base de donnée, rendant possible de tester l'unité de code considérée avec une couverture adéquate. L'extension que nous proposons est implémentée dans un outil et évaluée sur des échantillons de code mélangeant instructions Java et SQL. Notre approche est comparée avec les travaux similaires. Une série de directions de recherche sont identifiées et discutées pour rendre possible son utilisation à grande échelle.

Mots-clés. Test Logiciel, Programmation par Contraintes, Bases de Données.

Abstract. All human beings make mistakes. Programmers are human beings. Thus programmers make mistakes. This seemingly harmless syllogism can have disastrous consequences for the society. As larger parts of our economic and social life rely on computers, software bugs caused by programming mistakes do pose an increasing thread. An example which has left its mark in people's minds is the Ariane 5 rocket explosion in flight, due to an integer overflow in the onboard computer.

In current software engineering practice, testing is the primary approach to locate programming mistakes. Before going into production, software is thus tested in a set of real scenarios, in order to detect the bugs at best. Nevertheless, the concrete process enabling software testing in practice is complex, repetitive and costly. This is why researchers and industrialists have been working to automate it for long. The idea is notably that the computer helps the programmer to select automatically the most relevant test scenarios. An interesting technique to do so is symbolic execution, today in use at Nasa and Microsoft. Technically, given a code unit to test, symbolic execution enables the computer to generate test values for the input parameters, making it possible to cover a representative set of execution paths in the code.

The present work introduces an extension for symbolic execution, enabling one to use this approach in a new context: testing transactional and data-oriented code units. This kind of code units, frequent in practice, is characterized by a brief and targeted interaction with a relational database, storing large amounts of information in a structured way. Finding bugs in such a kind of code unit is critical, as it can avoid corrupting possibly very important data. In order to achieve this, our work goes back to the theoretical foundations of relational databases and proposes a relational symbolic execution mechanism for SQL. This enables generating automatically input data, including some valid content for the database, making it possible to test the considered code unit with an adequate coverage. The proposed extension is implemented within a tool and evaluated over code samples mixing Java and SQL statements. Our approach is compared with related work. A set of research directions to make it usable at a larger scale are identified and discussed.

Keywords. Software Testing, Constraint Programming, Databases.

Remerciements

Un très grand homme a dit un jour “l’union fera notre force, la division notre faiblesse”. Je voudrais profiter de ces lignes pour remercier toutes celles et ceux qui m’ont accompagné durant mes années de thèse. En particulier, je remercie...

Le professeur Wim Vanhoof et le professeur Jean-Luc Hainaut, mes promoteur et co-promoteur, pour avoir été présents quand cela était nécessaire, tout en me laissant la liberté de découvrir le chemin et d’apprendre à le tracer en toute indépendance. Dans la gestion quotidienne de la thèse, le professeur Wim Vanhoof pour sa grande capacité d’écoute et son ouverture d’esprit. Le professeur Pierre-Yves Schobbens pour nos discussions scientifiques plus qu’enrichissantes autour de ma cafetière. Les professeurs Vincent Englebert et Anthony Clève pour leurs idées, conseils et coups de pouce avisés. Le docteur Gilles Perrouin pour nos conversations passionnantes, sur la vie d’un chercheur et les solvers de contraintes. Le professeur Jef Wijzen et le docteur Alexander Serebrenik pour leurs remarques enrichissantes. L’ensemble des reviewers qui ont relu mes articles pour leurs commentaires pertinents et constructifs. L’ensemble du personnel administratif et technique de la faculté d’informatique et de l’université de Namur, pour tout le travail de l’ombre. Cédric Aerts, en particulier, pour la disponibilité du matériel de test.

Le professeur Pierre-Yves Schobbens et Xavier Devroey. Cela a été un plaisir de donner du fil à retordre aux étudiants avec vous (et, j’en suis sûr, leur avoir appris l’une ou l’autre chose par la même occasion). Le professeur Vincent Englebert, le docteur Aude Nguyen et toute l’équipe de la commission de l’enseignement facultaire, pour la qualité du travail réalisé ensemble. Le doyen Naji Habra, pour son leadership bienveillant, le vice-doyen Laurent Schumacher, pour son professionnalisme à toute épreuve et son sens de l’humour, ainsi que toutes celles et ceux qui donnent de leur temps et de leur énergie pour le bon fonctionnement de la faculté d’informatique. L’ensemble des gens avec qui j’ai eu le plaisir de collaborer à l’Université de Namur. Romain Demeyer et Céline Dandois qui ont été d’excellents collègues de labo. Le professeur Michaël Petit, pour le mille-pattes et tout ce que nous avons fait ensemble. Céline, Bastien, Guess, Seweryn et Anthony, ou plutôt, devais-je dire, Ophélie, Barnabé, Lisa, Raymond, le voisin et toute l’équipe d’*un Monde sans Voyelle*. Cela a été un grand moment.

Ma famille, mes amis, et en particulier Tonia, Gino, Stéphane, Anne-Lise et tous les membres de la troupe du *théâtre universitaire de Namur*. Toute l’équipe d’*écolo j*, où j’ai beaucoup appris, et en particulier Sara, Giuseppe et Nathalie Grandjean, pour ce que nous avons construit ensemble.

Toutes celles et ceux dont le nom ne tenait pas sur cette unique page, mais qui occupent toute leur place dans mon cœur.

Acknowledgments

A very great man said one day “we are only as strong as we are united, as weak as we are divided”. I would like to take advantage of these lines to thank all the ones who have been with me during my thesis years. In particular, I thank...

Professor Wim Vanhoof and professor Jean-Luc Hainaut, my supervisor and co-supervisor, for having been there when necessary, while leaving me the freedom to discover the way and to learn how to make my own one independently. Regarding the daily management of the thesis, professor Wim Vanhoof for his great ability to listen and his open-mindedness. Professor Pierre-Yves Schobbens for our more than enriching scientific discussions around my coffee maker. Professors Vincent Englebert and Anthony Clève for their shrewd ideas, advices and boosts. Doctor Gilles Perrouin for our exciting conversations, about a researcher’s life and constraint solvers. Professor Jef Wijsen and doctor Alexander Serebrenik for their enriching remarks. All the reviewers who have read my articles for their relevant and constructive comments. All the administrative and technical staff of the faculty of computer science and university of Namur, for all the work done in the shadows. Cédric Aerts, in particular, for the availability of the test hardware.

Professor Pierre-Yves Schobbens and Xavier Devroey. It has been my pleasure to give students a hard time with you (and, I am sure, learning them one or two things by the way). Professor Vincent Englebert, doctor Aude Nguyen and the whole team of the faculty’s teaching commission, for the quality of the work performed together. Dean Naji Habra, for his benevolent leadership, vice-dean Laurent Schumacher, for his unfailing professionalism and his sense of humour, as well as all those who spend their time and energy to ensure the smooth running of the faculty of computer science. All the people it has been my pleasure to collaborate with at the university of Namur. Romain Demeyer and Céline Dandois who have been excellent lab colleagues. Professor Michaël Petit, for the centipede and all what we have been doing together. Céline, Bastien, Guess, Seweryn and Anthony, or should I say, Ophélie, Barnabé, Lisa, Raymond, the neighbour and the whole team of *un Monde sans Voyelle*. It has been a great time.

My family, friends, and in particular, Tonia, Gino, Stéphane, Anne-Lise and all the members of the *théâtre universitaire de Namur*. The whole team of *écolo j*, where I have learnt quite a lot, and especially Sara, Giuseppe and Nathalie Grandjean, for all what we have built together.

All the ones whose name could not fit on this single page, but who have a full place in my heart.

Contents

Introduction	1
I SQL, Constraints and Test	5
1 Relational Databases	7
1.1 The Relational Model of a Database	9
1.1.1 Mathematical Concepts and Notation	9
1.1.2 A Relational Representation to Store Information	11
1.1.3 The Relational Algebra	14
1.1.4 The Relational Calculus	18
1.1.5 Uniqueness and Referential Integrity	22
1.2 Relational DBMS and SQL	24
1.2.1 Relational Database Management Systems	24
1.2.2 The SQL Language	25
1.2.3 SQL in Software Applications	35
1.3 Alternatives to Relational Databases	40
2 SAT, SMT and Relational Solvers	45
2.1 SAT Solvers	47
2.1.1 The SAT Problem	47
2.1.2 Complexity of SAT and Practical Applications	48
2.1.3 The DPLL Algorithm for SAT	48
2.1.4 Alternative Algorithms for SAT	53
2.2 SMT Solvers	55
2.2.1 The SMT Problem	55
2.2.2 The SMT-Lib Language	57
2.2.3 Complexity of SMT and Practical Applications	59
2.2.4 The DPLL(T) Algorithm for SMT	60
2.2.5 The Nelson-Oppen Method for Theory Combination	64
2.2.6 Handling Quantifiers	65
2.2.7 Challenges and Alternatives	72
2.3 Relational Solvers	74
2.3.1 Writing and Solving Relational Constraints	74
2.3.2 The Alloy Framework	74
2.3.3 From Alloy to SMT	79

3	Constraint-Based Software Testing	81
3.1	Software Testing	83
3.1.1	Principles and Automation	83
3.1.2	Testing Target and Objective	83
3.1.3	Alternative Approaches	84
3.2	The Constraint-Based Approach	85
3.2.1	Purpose and Principle	85
3.2.2	Symbolic Execution	86
3.2.3	Static and Dynamic Path Exploration	88
3.2.4	Recent Trends	92
3.2.5	Alternative Automatic Test Input Generation Techniques	95
4	Constraint-Based Testing of SQL Applications: a State of the Art	97
4.1	SQL Normalization into Native Code	99
4.2	Query Results as Input Arrays	100
4.3	Related Research over SQL Application Testing	102
4.3.1	Requirement and Opportunity Analysis	102
4.3.2	Selecting Inputs for Apps Working on an Existing Database	102
4.3.3	Code Coverage Criteria for SQL Applications	102
4.3.4	Code-Independent Test Database Generation	103
4.3.5	Other Testing Approaches and Objectives	105
II	Relational Testing of SQL Applications	107
5	Relational Testing: an Introduction	109
5.1	Motivations	111
5.2	Principle	113
5.3	Methodology	115
6	SimpleDB, a Core Language for Studying SQL Applications	117
6.1	Defining a Core Java/SQL for OLTP Use Cases	119
6.2	A Formal Definition for SimpleDB	120
6.2.1	SimpleDB Program	120
6.2.2	Database Schema	120
6.2.3	Method Signature and Body	122
7	A Relational Algorithm for Symbolic Execution	125
7.1	Principle	127
7.2	Alloy Version	128
7.2.1	Execution Example	128
7.2.2	Constraint Generation Rules	133
7.3	SMT-Lib Version	138
7.3.1	Execution Example	138
7.3.2	Constraint Generation Rules	144

8	Experimental Evaluation	149
8.1	Experimental Framework	151
8.1.1	A Tool for Relational Testing	151
8.1.2	Sample Code for Experimentation	151
8.2	Evaluating the Alloy and Z3 Solvers for Relational Testing	152
8.2.1	Performance over Commonly Crafted Samples	152
8.2.2	Practical Scalability Limits for Z3	154
8.3	Benchmarking Relational Testing against Normalization	158
8.4	Additional Experiments over Open-Source Code	158
9	Discussion	161
9.1	Synthesis of Research Contributions	163
9.2	Comparison with Related Work	163
9.2.1	SQL Normalization into Native Code	163
9.2.2	Query Results as Input Arrays	164
9.2.3	Qex	164
9.3	Threats to Validity	165
9.3.1	Internal Validity	165
9.3.2	External Validity	165
III	A Path for the Future of Relational Testing	167
10	From SimpleDB towards Full-Scale SQL Applications	169
10.1	Sample Code for Discussion	171
10.2	Integration with Constraint-Based Testing Tools	171
10.3	Handling Dynamically-Crafted SQL	173
10.4	Building and Solving Constraints for SQL	173
10.5	Detecting SQL Runtime Errors	174
10.6	Handling Non-Deterministic SQL	175
11	Possible Extensions and New Challenges	177
	Using Dedicated Code Coverage Criteria	177
	Taking Advantage of DDL and Business Use Cases	177
	Extracting Test Data from an Existing Database	177
	Minimizing Database Modifications between Tests	178
	Considering other Data Technologies	178
	Conclusion	181
	Bibliography	185
IV	Appendices	203
A	Nelson-Oppen Combination	205
A.1	Convex Σ -theories	205
A.2	Convex Nelson-Oppen Combination	205

B	Code Samples Used for Evaluation	211
B.1	SimpleDB Samples	211
B.1.1	SimpleDB Language Unit Testing	211
B.1.2	Realistic OLTP Methods	217
B.2	Samples Extracted from OpenSource Software	222
B.2.1	UnixUsage	222
B.2.2	RiskIt	224

Introduction

ON a single researcher scale, research is the process of building new knowledge, still unknown to mankind. As any building process, it requires strong foundations. Researchers must first develop a solid background in their general research field, together with an extensive mastering of the state of the art for the particular zone they investigate. These two elements are necessary both to identify genuine “holes” in human knowledge, deserving to be filled up by new research, and to benefit from the large amounts of conceptual material built by the research community, necessary to try plugging the identified holes. Plugging these holes is first a matter of researcher creativity, fed by that conceptual material they have accumulated, and driven by intuition, trials, errors,... and sometimes success. Plugging these holes is also a matter of a structured, rigorous and honest methodological approach, observing facts and (in)validating hypothesis, in an uncompromising quest for truth.

In order for human knowledge to keep up growing at a more general scale, the new amount of knowledge produced by single researchers must be made available at long term and in an organized way for the researcher community. This increases the knowledge actually and permanently available in the society and feeds thus the future work of other researchers. Communicating about their research outputs is thus an important task for researchers. Such a communication essentially occurs by the mean of ad-hoc institutions, like scientific journals and conferences, often run by the researchers themselves. Peer-review is the core mechanism of such an institutional scientific communication, where other researchers assess the quality of the communicated contribution, before allowing its publication. Constructive peer-review can be a strong driver for researchers to improve their research. All these elements cause research to be a deeply social activity, thus also driven by relational and political aspects. Dealing with competition, promotion and networking, but also building collaborations, leading sometimes even to friendship, are thus also important elements of a researcher’s life.

At the society level, the question of the purpose, means, limits and governance of research raises many ethical, ideological and political debates, together with practical issues, with whom researchers are directly concerned. Should it be allowed to research about everything, whatever the impact might be? Who should be responsible to decide which research topics must be funded or not? According to which criteria? Should researchers be funded to be free to decide on their own research objectives and agenda? What should be the link between research, industry and business? All these questions are examples of debates whose answers can have a strong impact over a researcher’s motivations, behavior as well as moral and material condition.

The current thesis manuscript essentially intends to describe a novel research work. First, it details the scientific background on which this research was built and enlightens the state of the art of the investigated research zone. Secondly, it identifies the holes in human knowledge filled by the research, discusses the scientific and societal interest to fill these holes, presents our methodological approach and details extensively the new knowledge we

have produced. Thirdly, it presents the new interesting holes to be filled which have been identified along the research, together with research intuitions to plug them. The peer-reviewed scientific papers published on this research work are mentioned where appropriate.

All human beings make mistakes. Programmers are human beings. Thus programmers make mistakes. This seemingly harmless syllogism can have disastrous consequences for the society. As larger parts of our economic and social life rely on computers, software bugs caused by programming mistakes do pose an increasing threat. An example which has left its mark in people's minds is the Ariane 5 rocket explosion in flight, due to an integer overflow in the onboard computer.

In current software engineering practice, testing is the primary approach to locate such programming mistakes. Before going into production, software is thus tested in a set of real scenarios, in order to detect the bugs at best. Nevertheless, the concrete process enabling software testing in practice is complex, repetitive and costly. This is why researchers and industrialists have been working to automate it for long, in order to improve both software cost and reliability. The idea is notably that the computer helps the programmer to select automatically the most relevant test scenarios. An interesting technique that can be employed to do so is symbolic execution. This approach, based on constraint programming, is today in use notably at Nasa and Microsoft. Technically, given a code unit to test, symbolic execution enables the computer to generate test values for the input parameters, covering a representative set of execution paths in the code.

The present work introduces an extension for symbolic execution, making it possible to use this approach in a new context: testing transactional and data-oriented code units. This kind of code unit, frequent in practice, mixes classical statements with SQL statements to perform a brief and targeted interaction with a relational database, the latter storing large amounts of information in a structured way. An example of such a code unit could be an application registering a patient in some hospital software system. This software system is supposed to be built around a relational database containing a large amount of structured information about patients, physicians, operations, bills, wages, etc. The application registering patients is a component of the whole software system, used when a patient is admitted to the hospital. It makes it possible to insert or update the patient's medical data in the database, so that they can be obtained later by the physicians who will heal the patient. Finding bugs in such a kind of a code unit is critical, as it can avoid corrupting possibly very important data. For illustrative purpose, let us consider that the patient's blood type gets incorrectly registered in the database by the application. If physicians use this corrupted piece of information later without any additional check, they may transfuse a blood of the wrong type to the patient, causing potentially serious damage.

In order to automate the testing of such transactional and data-oriented code units, our work goes back to the theoretical foundations of relational databases and proposes a relational symbolic execution mechanism for SQL. This technique, called relational testing, enables generating automatically input data, including some valid content for the database, making it possible to test the considered code unit with an adequate coverage. Relational testing is implemented within a tool and evaluated over code samples mixing Java and SQL statements. The approach is compared with related work. A set of research directions to make it usable at a larger scale are identified and discussed.

The text is divided into three main parts.

The first part, involving chapters 1 to 4, details the scientific background within which relational testing was developed, together with the current state-of-the-art about testing applications interacting with a database. Chapter 1 offers a general discussion about relational databases and SQL. In chapter 2, we provide an operational description of the architecture of constraint solvers. Chapter 3 details the current state of art in using symbolic execution for constraint-based testing. Chapter 4 presents other test generation approaches that handle SQL, together with a more general review of related work for relational testing.

The second part, involving chapters 5 to 9, is a detailed discussion of the relational testing technique developed in this thesis. Chapter 5 introduces our motivations and methodology for developing relational testing. It also explains the principle of the approach, relying on the elements introduced in chapter 1, 2, 3 and 4. Following the methodological approach that we used for our research, chapters 6 to 9 presents the relational testing approach as well as its experimental evaluation and comparison with related work. Threats to validity are discussed at the end of chapter 9.

The third and last part, involving chapter 10 and 11, details some research directions for growing the relational testing technique towards a better scope and scalability. Chapter 10 sketches a path to enlarge the panel of the data-driven applications handled by relational testing. Following this path could enable one to remove several of the threats to validity elucidated in chapter 9 and make an evaluation of relational testing in an industrial context possible. Chapter 11 briefly relates, in a prospective way, some possible extensions and new challenges around relational testing.

Part I

SQL, Constraints and Test

“If I have seen further, it is by standing on the shoulders of Giants.”
ISAAC NEWTON

Chapter 1

Relational Databases

Contents

1.1	The Relational Model of a Database	9
1.1.1	Mathematical Concepts and Notation	9
1.1.2	A Relational Representation to Store Information	11
1.1.3	The Relational Algebra	14
1.1.4	The Relational Calculus	18
1.1.5	Uniqueness and Referential Integrity	22
1.2	Relational DBMS and SQL	24
1.2.1	Relational Database Management Systems	24
1.2.2	The SQL Language	25
1.2.3	SQL in Software Applications	35
1.3	Alternatives to Relational Databases	40

APPLICATIONS executed by computers are formal procedures describing sequences of operations processing information about the world. Most of the magic of computing holds in the speed of computers and the ubiquity of computer networks, which enable this information about the world to be analyzed, transformed and used quickly and in huge quantities, as well as shared across the planet at the speed of light. As a consequence, a core design problem when building software applications is how to represent and manipulate the information to be processed by the application. In the simplest cases, this information can be entirely captured by a few scalar values loaded in the application's variables. In the hardest cases, the information is made of large quantities of inter-related facts and must be permanently stored and be available for retrieval and modification to a number of applications. Such cases require to use efficient *DataBase Management Systems* (DBMSs), i.e. software systems providing applications with a shared access to an organized and persistent representation of the information, called a *database*.

The *relational model of a database*, developed from an original proposition by Codd in 1970, introduces a standardized and theoretical "design pattern" for building a database and enabling the manipulation of its content. It provides common conceptual tools to structure, offer access and enable modification of information. A *relational database* is a database built and accessed according to the design pattern described by the relational model. A *relational DBMS* is a software utility that makes it possible to create relational databases and manipulate their content, implementing in practice the conceptual tools provided by the relational model. Relational DBMSs have become the de facto standard dominating the database market for a number of decades, despite the emergence of several *contending approaches*. At the core of all the relational DBMSs stands the *Structured Query Language* (SQL), a language standardized by ISO for creating relational databases and manipulating their content, which is strongly tight to the mathematical concepts of the relational model. DBMSs typically provide standard *transactional consistency and reliability guaranties* for the executed SQL code.

This chapter discusses the relational model (section 1.1), presents relational DBMSs, the SQL language and their integration with software applications (section 1.2), and finally concludes by considering alternative approaches for database design (section 1.3). The text offers an original synthesis from several sources, including book references in the domain [Abiteboul et al., 1995, Hainaut, 2012, Connolly and Begg, 2005]. References to these books will not be systematically repeated in the chapter. The proposed synthesis is essentially specific as it employs the concept of relational variables [Darwen and Date, 1995] as a metaphor to explain some core principles of the relational model and because it emphasizes the links between the SQL language, relational operators and first-order logic. By showing how SQL code can be compiled into a language made of relational operators and variables, equivalent to first-order logic, this chapter lays the conceptual foundations for the SQL symbolic execution algorithms proposed in this thesis. By bringing the main non relational part of SQL out, this chapter paves the way for introducing the considered extensions of these algorithms.

1.1 The Relational Model of a Database

The relational model of a database is a theoretical pattern to organize a database, i.e. to build a permanent and structured store for information about a given domain of interest in the world, which offers access and modification of this information. The design pattern offered by the relational model is generic, as it can be used to build a specific database whatever the domain of interest is about. The relational model was originally proposed by Codd in a seminal paper written in 1970 [Codd, 1970] and subsequently extended in many ways, as the theoretical and practical research on the topic grew in importance. Nowadays, it can be synthetically described as a framework without definite borders, but composed of three main conceptual tools for a standardized database design:

- A mathematical abstraction to build a storable representation of information, whatever this information is about, using the simple concept of relation.
- Two equivalent mathematical languages that respectively offer an imperative and declarative mean to manipulate the stored relational representation of information: the relational algebra and the relational calculus.
- The concepts of integrity and normalization, which complement the principle of a relational representation of information with a mathematical foundation to prevent the representation of invalid or redundant information.

In the remainder of this section, we show how one can take advantage of these standardized design tools to build a database. We consider, without loss of generality, the example of a bookshop as an illustrative domain of interest to represent. Basically, our purpose is to build conceptually a database storing and providing access to information about the resources available in the bookshop. This database is intended to be used as a storage component in the bookshop's management software. In subsection 1.1.1, we remind the necessary mathematical concepts and notations. In subsection 1.1.2, we build a storable representation of the information concerning the bookshop, using the relational abstraction advocated by the relational model. Then, we show how the information represented in this way can be manipulated using the relational algebra (subsection 1.1.3) and calculus (subsection 1.1.4). Finally, we present the two most common integrity rules used in relational modeling, i.e. uniqueness and referential integrity, and show how they apply to the bookshop case (subsection 1.1.5). The question of normalization is out of the scope considered in this work, but the interested reader can notably find detailed information over this topic in [Codd, 1971, Garcia-Molina et al., 2008].

1.1.1 Mathematical Concepts and Notation

The relational model of a database is based on the simple mathematical concepts of sets, relations and predicates. As an introduction to the relational model, we briefly remind here the formal definition and notation for these three concepts and emphasize the existing links between them.

Definition 1.1. A *set* is a collection of distinct objects.

The word object is used here as a generic noun for everything that exists, including both conceptual and physical items. If an object e is an element of S , we write $e \in S$. The number of elements in a given set is called the *cardinality* of the set.

A typical example of set is $\mathbb{Z}$, the infinite set of integers. The integers 0, 1 and 2 are some among the elements of $\mathbb{Z}$ and we can write $0 \in \mathbb{Z}$, $1 \in \mathbb{Z}$ and $2 \in \mathbb{Z}$. The cardinality of the finite integer set $\{0, 1, 2\}$, denoted by $\#\{0, 1, 2\}$, is 3.

Definition 1.2. A **subset** B of a set S is a set whose elements are also elements of S .

If B is a subset of S , we write $B \subseteq S$. It should be noted that this definition allows the empty set and the set S itself to be subsets of S .

As an example, the set $\{0, 1, 2\}$ is a subset of set $\mathbb{Z}$ and we write $\{0, 1, 2\} \subseteq \mathbb{Z}$.

Definition 1.3. A boolean function P defined on a set S is called a **predicate** on S .

One can easily establish the equivalence lemma between subsets and predicates:

Lemma 1.1. With each predicate P_B defined on a set S , one can always associate a subset B of S containing all the elements of S for which P_B is true. Conversely, with each subset B of a set S , one can always associate the predicate P_B on S , which returns true for each element of S that is also an element of B and false for all the other elements of S .

Definition 1.4. The predicate P_B equivalent to the subset B is called the **characteristic function** of B .

This formal bijection between the subsets and predicates defined on a set S allows a subset B of S to be represented using the following predicative notation:

$$\{s \in S \mid P_B(s)\}.$$

For example, the characteristic function of the subset $\{0, 1, 2\}$ of $\mathbb{Z}$ is the predicate $P_{\{0,1,2\}}$:

$$P_{\{0,1,2\}} : \mathbb{Z} \mapsto \{\text{true}, \text{false}\} : P_{\{0,1,2\}}(z) = (z \geq 0 \wedge z \leq 2)$$

and $\{0, 1, 2\}$ can then be represented using the following predicative notation:

$$\{z \in \mathbb{Z} \mid z \geq 0 \wedge z \leq 2\}.$$

Definition 1.5. The **power set** of a set S is the set of all subsets of S .

We write $\mathcal{P}(S)$ the power set of a set S . As an example, the power set of set $\{0, 1, 2\}$ is:

$$\mathcal{P}(\{0, 1, 2\}) = \{\emptyset, \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\}\}$$

Definition 1.6. A **tuple** built from a list of n sets is a list of n elements taken in order from each of the sets.

Definition 1.7. The **cartesian product** of a list of sets is the set of all the possible tuples that can be built from this list of sets.

We write $S_1 \times \cdots \times S_n$ the cartesian product of the list of sets $S_1, \cdots, S_n$.

As an example, we can build the eight tuples that constitute the cartesian product of the list of three sets $\{0, 1\}$, $\{1, 2\}$ and $\{2, 3\}$:

$$\{0, 1\} \times \{1, 2\} \times \{2, 3\} = \{ (0, 1, 2), (0, 1, 3), (0, 2, 2), (0, 2, 3), \\ (1, 1, 2), (1, 1, 3), (1, 2, 2), (1, 2, 3) \}$$

Definition 1.8. A *relation* R defined over a list of sets is a set of tuples built from this list of sets.

Each set in the list of sets on which a relation is built and its associated position in the list defines one of the so-called *attributes* of the relation. Each of the attributes of a relation is sometimes given a proper distinguishing name. For example, the value of the second attribute, called attribute B, for a tuple $(1, 2, 3)$ extracted from a given relation, is 2. The number of attributes for a given relation is called the *arity* of the relation. In this work, we will only consider relations with a finite arity and cardinality.

By definitions 1.2 and 1.8, we have the following lemma:

Lemma 1.2. A relation R defined over a list of sets is a subset of the cartesian product of this list of sets.

As a consequence of lemmas 1.1 and 1.2, we have the following fundamental result, establishing a formal equivalence between relations and predicates:

Result 1.1 (Predicate-Relation Equivalence). *With each predicate P_R defined on the cartesian product of a list of sets, one can associate a relation R defined over this list of sets containing all the tuples for which P_R is true. Conversely, with each relation R defined over a list of sets, one can associate the predicate P_R on the cartesian product of this list of sets, which returns true for each tuple that is an element of R and false for the other tuples.*

A relation R defined over the list of sets $S_1, \dots, S_n$ can then be represented using the predicative notation introduced previously:

$$\{(s_1, \dots, s_n) \in S_1 \times \dots \times S_n \mid P_R(s_1, \dots, s_n)\}.$$

For example, the characteristic function of the relation $\{(0, 1, 2), (1, 2, 3)\}$ containing two tuples built from the list of three sets $\{0, 1\}$, $\{1, 2\}$ and $\{2, 3\}$ is the predicate $P_{\{(0,1,2),(1,2,3)\}}$:

$$\begin{aligned} P_{\{(0,1,2),(1,2,3)\}} : \{0, 1\} \times \{1, 2\} \times \{2, 3\} &\mapsto \{true, false\} : \\ P_{\{(0,1,2),(1,2,3)\}}(s_1, s_2, s_3) &= ((s_2 = s_1 + 1) \wedge (s_3 = s_2 + 1)) \end{aligned}$$

and $\{(0, 1, 2), (1, 2, 3)\}$ can then be represented using the following predicative notation:

$$\{(s_1, s_2, s_3) \in \{0, 1\} \times \{1, 2\} \times \{2, 3\} \mid (s_2 = s_1 + 1) \wedge (s_3 = s_2 + 1)\}.$$

Finally, by definition 1.5 and lemma 1.2 we have the following result:

Result 1.2. *The power set of the cartesian product of a list of sets is the set of all the possible relations that can be defined over this list of sets.*

1.1.2 A Relational Representation to Store Information

A Representing the State of the World by a set of Relations

The relational model proposes a generic mathematical abstraction that enables building a storable representation of information. In concrete terms, the relational model suggests to represent the current state of the domain of interest by a set of relations. These relations can be easily stored and manipulated by a computer as they can be encoded as arrays,

whose rows are tuples and column names are attribute names. In this subsection, we demonstrate how the relational model makes it possible to represent the state of the bookshop's resources as a set of two relations: the **Shelf** and **Copy** relations.

Let us suppose that the bookshop's resources are composed of copies of books arranged into a number of shelves. Each book copy is tagged by a unique barcode and each shelf can be distinguished from the other ones using a unique shelf's number posted on the shelf, together with a more detailed description of the shelf's content, called the shelf's label. First, the relational model classifies all the items of information possibly relevant to describe the resources of the bookshop into five different types, which are, mathematically, sets of objects:

Identifiers = the set of all the shelf numbers that can be used in the bookshop
Labels = the set of all the shelf labels that can be used in the bookshop
Barcodes = the set of all book copy barcodes that can be used in the bookshop
Titles = the set of all possible book titles
Authors = the set of all possible book authors

Secondly, the relational model defines two predicates P_{Shelf} and P_{Copy} over these five sets to describe in a formal way the resources currently present in the bookshop. Each shelf in the bookshop can be distinctly represented by a tuple containing the shelf's identifier and the shelf's label. The predicate P_{Shelf} will return true for each tuple $(identifier, label)$ defined on **Identifiers** $\times$ **Labels** that represents a shelf that currently stands in the bookshop:

$P_{Shelf} : \mathbf{Identifiers} \times \mathbf{Labels} \mapsto \{true, false\} :$
 $P_{Shelf}(i, l) =$ The bookshop currently contains a shelf with identifier i and label l

By wandering in the bookshop and examining each of the shelves standing there, one can list all the tuples composing the relation **Shelf** of which P_{Shelf} is the characteristic function. For example, such a relation could be, for a bookshop containing two shelves:

Relation Shelf	
Identifier	Label
13	Fantasy
14	Horror

Similarly, each book copy in the bookshop can be distinctly represented by a tuple containing the copy's barcode, title, author and the identifier of the shelf where the copy stands. The predicate P_{Copy} will return true for each tuple $(barcode, title, author, identifier)$ defined on **Barcodes** $\times$ **Titles** $\times$ **Authors** $\times$ **Identifiers** that represents a book copy currently available in the bookshop:

$P_{Copy} : \mathbf{Barcodes} \times \mathbf{Titles} \times \mathbf{Authors} \times \mathbf{Identifiers} \mapsto \{true, false\} :$
 $P_{Copy}(b, t, a, n) =$ The bookshop currently owns a copy with barcode b
of a book with title t , written by author a and
which is stored in the shelf with identifier i

By wandering in the bookshop and examining each of the book copies standing there, one can list all the tuples composing the relation **Copy** of which P_{Copy} is the characteristic function. For example, such a relation could be, for a bookshop containing four books:

Relation Copy			
Barcode	Title	Author	Identifier
45536	Harry Potter and the Deathly Hallows	J.K. Rowling	13
45535	Harry Potter and the Deathly Hallows	J.K. Rowling	13
56563	The Return of the King	J.R.R. Tolkien	13
87567	The Shining	Stephen King	14

The **Shelf** and **Copy** relations are a representative example of the formal and structured representation of the known state of a domain of interest that can be built using the relational model.

B Relational Variables

A database does not only offer an information store, but also provides abilities to query and update the stored information. Queries enable one to extract useful facts from the stored information. For example, the bookshop's employees will typically use the bookshop's software to know the list of book copies available for a given author and the location of these books among the shelves. This information can be extracted from the **Copy** relation and encoded into the following new relation (for author J.K. Rowling):

Relation CopyFromRowling		
Barcode	Title	Shelf's Identifier
45536	Harry Potter and the Deathly Hallows	13
45535	Harry Potter and the Deathly Hallows	13

Updates enable one to modify the stored information in order to faithfully reflect the changes occurring in the domain of interest. The **Shelf** and **Copy** relations are indeed limited to represent a static snapshot of the state of the world at a precise moment in time, since the domain of interest can change dynamically as time goes by. For example, if a bookshop's client buys the Stephen King's book from the bookshop, this fact will thus be saved by the bookshop's software and the books available in the bookshop will then be represented by the new relation **Copy'**:

Relation Copy'			
Barcode	Title	Author	Identifier
45536	Harry Potter and the Deathly Hallows	J.K. Rowling	13
45535	Harry Potter and the Deathly Hallows	J.K. Rowling	13
56563	The Return of the King	J.R.R. Tolkien	13

The generic mechanism proposed by the relational model to query and update the stored information is similar to the one proposed by programming languages to use and update variables, and a relational database (i.e. a database built according to the relational model) can be seen as a set of variables, typed as relations, which can be manipulated using relational expressions.

In many programming languages, a typed variable is a container for a value of a fixed type. Variables are manipulated using expressions, i.e. constructs composed of variables, constant values and operators, which represent values. In a program, using some variables as part of an expression makes it possible to compute a new value from these variables. This newly computed value can then be used in turn to set or update the value a program

variable. For example, let us consider the following assignment that could be extracted from the air conditioning component of the bookshop software:

```
wantedTemperatureRise = expectedTemperature - bookshopTemperature;
```

It computes the wanted temperature rise to be performed by the air conditioning system as the value of the right expression, involving the `expectedTemperature` and `bookshopTemperature` real variables, as well as the minus operator, and puts this expression's value into the `wantedTemperatureRise` real variable on the left.

Similarly, the database built according to the relational model for the bookshop is not composed of the two static relations `Shelf` and `Copy`, but of the two *relational variables* `shelf` and `copy`. The variable `shelf` is typed as a relation over **Identifiers** $\times$ **Labels** and contains the relation describing the shelves that are currently present in the bookshop. The variable `copy` is typed as a relation over **Barcodes** $\times$ **Titles** $\times$ **Authors** $\times$ **Identifiers** and contains the relation describing the book copies that are currently present in the bookshop. At any moment, these variables can be used in relational expressions to build new relations which answer meaningful queries over the database content. When the bookshop's resources evolve, the content of these two variables is consequently updated, by computing the updated values from the older ones using relational expressions. The relational model enables writing such relational expressions by introducing the *relational algebra*, a set of operators defined on relations, which makes one able to build new relations from other ones. The relational algebra, as a universal language acting on relations, can be used to manipulate information in any relational database made of a set of relational variables, whatever these relational variables represent.

1.1.3 The Relational Algebra

A Relational Completeness

In this subsection, we discuss the relational algebra and show how it can be employed to compute new relations from the relational variables modeling a given domain of interest. In concrete terms, we present and demonstrate the five relational operators (selection, projection, product, union and difference) that form the minimal set of operators of the relational algebra. In the literature, some additional relational operators have been defined for convenience purpose, like intersection, joins or divisions. All these operators can be expressed in terms of the five fundamental operators. Any relation manipulation language that has at least the same expressive power as the relational algebra (i.e. which can simulate any combined action of its five fundamental operators) is said to be *relational complete* [Connolly and Begg, 2005]. It should be noted that to keep the relational algebra computable in a finite time (which is a desirable property in a database), its operators are only defined over finite relations.

B Relational Operators

B.1 Selection The selection operator σ receives a relation as input and builds an output relation by *selecting* a subset of the tuples from the input relation. Supposing that the input relation is defined over the list of sets $S_1, \dots, S_n$, the tuples are selected by defining a predicate

$$P : (S_1 \times \dots \times S_n) \mapsto \{true, false\}$$

over the cartesian product of these sets and by populating the output relation with the input tuples that make the predicate true (the predicate is thus the characteristic function of the selected subset). The defined predicate cannot be any predicate, but is limited to predicates establishing either that two attribute values in the tuple are equal or that one attribute value in the tuple is equal to a given constant value. Formally, selection can be defined using the predicative notation of the produced relation:

$$\begin{aligned} \sigma_P : \mathcal{P}(S_1 \times \cdots \times S_n) &\mapsto \mathcal{P}(S_1 \times \cdots \times S_n) : \\ \sigma_P(R) &= \{s \in (S_1 \times \cdots \times S_n) \mid (s \in R) \wedge P_X(s)\} \end{aligned}$$

As an example, one can use selection to list all the book copies written by J.K. Rowling from the **Copy** relation:

$$\sigma_{P_{Rowling}}(\text{Copy}) = \{(45536, \text{Harry Potter and the Deathly Hallows, J.K. Rowling, 13}), \\ (45535, \text{Harry Potter and the Deathly Hallows, J.K. Rowling, 13})\}$$

using predicate $P_{Rowling}$ defined as:

$$\begin{aligned} P_{Rowling} : (\mathbf{Barcodes} \times \mathbf{Titles} \times \mathbf{Authors} \times \mathbf{Identifiers}) &\mapsto \{true, false\} \\ P_{Rowling}((b, t, a, i)) &= (a = \text{"J.K. Rowling"}) \end{aligned}$$

B.2 Projection The projection operator π receives a relation as input and builds an output relation by *deleting*, *duplicating* and/or *reordering* the attributes of the input relation.

Definition 1.9. A **projection** is a function $p : 1 \cdots k \mapsto 1 \cdots n$, so that, given a list $S = S_1, \dots, S_n$, the projection generates a new list $M_1, \dots, M_k$ with

$$\forall i \in 1 \cdots k, M_i = S_{p(i)}$$

For example, the list of four sets **Barcodes**, **Titles**, **Authors**, **Identifiers** admits notably the following projections:

$\emptyset$	Barcodes
Titles	Authors
Identifiers	Barcodes, Titles
Authors, Barcodes	Identifiers, Identifiers
Identifiers, Titles, Identifiers	Authors, Titles, Barcodes
Identifiers, Identifiers, Identifiers	Barcodes, Titles, Authors, Identifiers
Authors, Identifiers, Titles, Barcodes, Identifiers, Identifiers	

If a list of sets $M_1, \dots, M_k$ is the list generated by the projection p when applied over the list of sets $S_1, \dots, S_n$, we can define the projection of a relation built from $S_1, \dots, S_n$ over $M_1, \dots, M_k$ using the predicative notation of the produced relation:

$$\begin{aligned} \pi_{(M_1, \dots, M_k)} : \mathcal{P}(S_1 \times \cdots \times S_n) &\mapsto \mathcal{P}(M_1 \times \cdots \times M_k) : \\ \pi_{(M_1, \dots, M_k)}(R) &= \{m \in (M_1 \times \cdots \times M_k) \mid (\exists s \in R, \forall i \in 1 \cdots k, m_i = s_{p(i)})\} \end{aligned}$$

As an example, we can use the projection operator over the **Copy** relation to list, for each author, the identifiers of all the shelves containing copies of books written by this author:

$$\pi_{(\mathbf{Authors}, \mathbf{Identifiers})}(\text{Copy}) = \{(\text{J.K. Rowling, 13}), (\text{J.R.R. Tolkien, 13}), (\text{Stephen King, 14})\}$$

B.3 Product The product operator $\times$ receives two relations as inputs and builds each of the tuples of the output relation by *concatenating* one of the tuples from the second input relation at the end of one of the tuples of the first input relation. Formally, the product operator can be defined using the predicative notation of the produced relation:

$$\begin{aligned} \times : (\mathcal{P}(A_1 \times \dots \times A_n) \times \mathcal{P}(B_1 \times \dots \times B_k)) &\mapsto \mathcal{P}(A_1 \times \dots \times A_n \times B_1 \times \dots \times B_k) : \\ R^A \times R^B &= \{(a_1, \dots, a_n, b_1, \dots, b_k) \in (A_1 \times \dots \times A_n \times B_1 \times \dots \times B_k) \\ &\quad | (a_1, \dots, a_n) \in R^A \wedge (b_1, \dots, b_k) \in R^B\} \end{aligned}$$

For example, we can build the product of the relations **Copy** and **Shelf**:

$$\begin{aligned} \text{Copy} \times \text{Shelf} = \\ \{ &(45536, \text{Harry Potter and the Deathly Hallows, J.K. Rowling, 13, 13, Fantasy}), \\ &(45535, \text{Harry Potter and the Deathly Hallows, J.K. Rowling, 13, 13, Fantasy}), \\ &(56563, \text{The Return of the King, J.R.R. Tolkien, 13, 13, Fantasy}), \\ &(87567, \text{The Shining, Stephen King, 14, 13, Fantasy}), \\ &(45536, \text{Harry Potter and the Deathly Hallows, J.K. Rowling, 13, 14, Horror}), \\ &(45535, \text{Harry Potter and the Deathly Hallows, J.K. Rowling, 13, 14, Horror}), \\ &(56563, \text{The Return of the King, J.R.R. Tolkien, 13, 14, Horror}), \\ &(87567, \text{The Shining, Stephen King, 14, 14, Horror}) \} \end{aligned}$$

B.4 Union and Difference The union operator $\cup$ and the difference operator $-$ are simply set union and set difference applied to relations. These operators can be applied on two relations if and only if these two relations have the same arity. For example, the union and difference operators can be applied over the two relations

$$R1 = \{(\text{J.K. Rowling, Crookshanks, Harry Potter and the Goblet of Fire}), \\ (\text{J.R.R. Tolkien, Smaug, The Hobbit})\}$$

$$R2 = \{(\text{J.K. Rowling, Crookshanks, Harry Potter and the Goblet of Fire}), \\ (\text{Wim Vanhoof, Tibby, Harry Potter and the Order of the Phoenix})\}$$

producing the two following relations:

$$\begin{aligned} R1 \cup R2 = \\ \{ &(\text{J.K. Rowling, Crookshanks, Harry Potter and the Goblet of Fire}) \\ &(\text{J.R.R. Tolkien, Smaug, The Hobbit}) \\ &(\text{Wim Vanhoof, Tibby, Harry Potter and the Order of the Phoenix}) \} \end{aligned}$$

$$R1 - R2 = \{(\text{J.R.R. Tolkien, Smaug, The Hobbit})\}$$

Formally, the union and difference operators can be defined using the predicative notation of the produced relation:

$$\begin{aligned} \cup : (\mathcal{P}(A_1 \times \dots \times A_n) \times \mathcal{P}(B_1 \times \dots \times B_k)) &\mapsto \mathcal{P}(S_1 \times \dots \times S_n) : \\ R^A \cup R^B &= \{s \in (S_1 \times \dots \times S_n) \mid s \in R^A \vee s \in R^B\} \end{aligned}$$

$$\begin{aligned} - : (\mathcal{P}(A_1 \times \dots \times A_n) \times \mathcal{P}(B_1 \times \dots \times B_k)) &\mapsto \mathcal{P}(S_1 \times \dots \times S_n) : \\ R^A - R^B &= \{s \in (S_1 \times \dots \times S_n) \mid s \in R^A \wedge \neg(s \in R^B)\} \end{aligned}$$

C Manipulating Variables using Operators

We can now show how the relational algebra can be used to manipulate the relational variables **shelf** and **copy**, composing the bookshop's relational database. Let us suppose that these variables have been respectively initialized by the example relations **Shelf** and **Copy**, which represent the current ressources available in the bookshop:

shelf = **Shelf**;
copy = **Copy**;

When a client enters the bookshop and asks “Excuse me, at which shelf's label can I find the Stephen King's books?”, the bookshop's software can extract the answer to this question (“All the available Stephen King's books are in shelves labelled with Horror, Sir!”) from the **shelf** and **copy** variables as the singleton relation $\{(\text{Horror})\}$ computed using the following relational expression and saved in a **labelKing** temporary relational variable:

$$\text{labelKing} = \pi_{\mathbf{Labels}}(\sigma_{P_{\text{SameIdentifier}}}(\sigma_{P_{\text{King}}}(\text{copy} \times \text{shelf})))$$

where the predicates $P_{\text{SameIdentifier}}$ and P_{King} are defined as:

$$\begin{aligned} &P_{\text{SameIdentifier}} : \\ &(\mathbf{Barcodes} \times \mathbf{Titles} \times \mathbf{Authors} \times \mathbf{Identifiers}_1 \times \mathbf{Identifiers}_2 \times \mathbf{Labels}) \mapsto \{true, false\} \\ &P_{\text{SameIdentifier}}((b, t, a, i_1, i_2, l)) = (i_1 = i_2) \end{aligned}$$

$$\begin{aligned} &P_{\text{King}} : (\mathbf{Barcodes} \times \mathbf{Titles} \times \mathbf{Authors} \times \mathbf{Identifiers}) \mapsto \{true, false\} \\ &P_{\text{King}}((b, t, a, i)) = (a = \text{“Stephen King”}) \end{aligned}$$

If the client finds and buys the only Stephen King's book copy available in the bookshop (identified by the barcode 87567), the content of the **copy** variable should be updated to reflect the change in the bookshop's ressources. This can be done using the following relational expression:

$$\text{copy} = \text{copy} - (\sigma_{P_{87567}}(\text{copy}));$$

resulting in the **copy** variable to contain the **Copy'** relation, defined previously. The predicate P_{87567} is defined here as:

$$\begin{aligned} &P_{87567} : (\mathbf{Barcodes} \times \mathbf{Titles} \times \mathbf{Authors} \times \mathbf{Identifiers}) \mapsto \{true, false\} \\ &P_{87567}((b, t, a, i)) = (b = 87567) \end{aligned}$$

Finally, if the bookshop receives an new book copy to sell (for example, a copy of ‘A Game of Thrones’ by George R. R. Martin with barcode 98215 to store in the shelf number 13), the content of the **copy** variable should also be updated to reflect this change in the bookshop's ressources. This can be done using the union operator:

$$\text{copy} = \text{copy} \cup \{(98215, \text{A Game of Thrones}, \text{George R. R. Martin}, 13)\}$$

resulting in the **copy** variable to contain the following relation:

$$\begin{aligned} &\{(45536, \text{Harry Potter and the Deathly Hallows}, \text{J.K. Rowling}, 13), \\ &\quad (45535, \text{Harry Potter and the Deathly Hallows}, \text{J.K. Rowling}, 13), \\ &\quad (56563, \text{The Return of the King}, \text{J.R.R. Tolkien}, 13), \\ &\quad (98215, \text{A Game of Thrones}, \text{George R. R. Martin}, 13)\} \end{aligned}$$

1.1.4 The Relational Calculus

A The Codd's Theorem

The predicate-relation equivalence introduced at page 11 establishes that the concepts of predicate and relation are equivalent. When Codd presented the relational algebra as a part of the relational model [Codd, 1970], logicians had already been writing logical formulas linking predicates for a long time, using the language of *first-order logic*. As a consequence, Codd established an equivalence between the relational algebra and the *relational calculus*, a subset of first-order logic, showing that any relational operator applied over relations can be described by an equivalent calculus formula linking the corresponding characteristic functions, and conversely. This equivalence, known as Codd's Theorem, provides us with two equivalent universal languages, either imperative (relational algebra) or declarative (relational calculus), to query and update the relational variables composing any relational database. In this subsection, we recall formally the syntax and semantic of first-order logic and discuss the translation from the relational algebra to the relational calculus.

B First-Order Logic

B.1 Syntax: Formula and Predicate First-order logic ([Abiteboul et al., 1995] in the context of databases) can be described as a formal language that enables one to define logical formulas combining predicates with logical connectors and quantifiers. A sample sentence in such a language, demonstrating most of its syntactic features, would be

```
PREDICATE P ON  $S_1 \times S_2$ ;  
VARIABLE x ON  $S_1 \times S_2$ ;  
FORMULA  $\exists x, P(x)$ ;  
FUNCTION F FROM  $S_3$  TO  $S_1 \times S_2$ ;  
VARIABLE y ON  $S_3$ ;  
CONSTANT c ON  $S_1 \times S_2$ ;  
CONSTANT true ON B;  
VARIABLE p ON B;  
FORMULA  $(true \wedge p \wedge P(F(y))) \implies \neg(F(y) = c)$ ;
```

where two logical formulas are defined (at the lines prefixed by the keyword **FORMULA**). Every symbol used in a formula is declared before being used. A predicate symbol declaration (line prefixed by the **PREDICATE** keyword) provides the symbols representing the set on which the predicate is defined. For example, the predicate represented by the symbol P is declared here to be defined over the cartesian product of the two sets represented by the symbols S_1 and S_2 . Declarations of variable symbols (**VARIABLE** keyword) and of constant symbols (**CONSTANT** keyword) are similar. Declarations of function symbols (**FUNCTION** keyword) provide symbols representing the domain and codomain sets of the represented function.

We define now formally the syntax of first-order logic using a BNF grammar. The chosen notation is standard but can include some additional meta-symbols: $\{...\}$ (grouping), $?$ (zero or one times), $*$ (zero or more times) and $^+$ (one or more times).

A sentence in the language of first-order logic, called a *theory*, is a list of interleaved declarations and formulas:

$$\langle theory \rangle ::= \{ \langle declaration \rangle ; \mid \mathbf{FORMULA} \langle formula \rangle ; \}^*$$

A declaration is either a predicate, a function, a variable or a constant symbol declaration:

$$\begin{aligned} \langle \text{declaration} \rangle &::= \text{PREDICATE } \langle \text{symbol} \rangle \text{ ON } S_1 \times \cdots \times S_n \\ &| \text{FUNCTION } \langle \text{symbol} \rangle \text{ FROM } S_1 \times \cdots \times S_n \text{ TO } T_1 \times \cdots \times T_m \\ &| \text{VARIABLE } \langle \text{symbol} \rangle \text{ ON } S_1 \times \cdots \times S_n \\ &| \text{CONSTANT } \langle \text{symbol} \rangle \text{ ON } S_1 \times \cdots \times S_n \end{aligned}$$

A formula is either a predicate application or a combination of predicate applications linked by logical connectors and quantifiers:

$$\begin{aligned} \langle \text{formula} \rangle &::= \langle \text{predicate} \rangle && \text{(Predicate Application)} \\ &| \neg \langle \text{formula} \rangle && \text{(Logical Not)} \\ &| \langle \text{formula} \rangle \wedge \langle \text{formula} \rangle && \text{(Logical And)} \\ &| \langle \text{formula} \rangle \vee \langle \text{formula} \rangle && \text{(Logical Or)} \\ &| \langle \text{formula} \rangle \implies \langle \text{formula} \rangle && \text{(Implication)} \\ &| \langle \text{formula} \rangle \iff \langle \text{formula} \rangle && \text{(Equivalence)} \\ &| \forall \langle \text{symbol} \rangle, \langle \text{formula} \rangle && \text{(Universal Quantifier)} \\ &| \exists \langle \text{symbol} \rangle, \langle \text{formula} \rangle && \text{(Existential Quantifier)} \\ &| (\langle \text{formula} \rangle) \end{aligned}$$

A predicate application is either a predicate symbol followed by the list of arguments on which the predicate is applied or the application of the equality predicate between two arguments, or a boolean variable/constant symbol:

$$\begin{aligned} \langle \text{predicate} \rangle &::= \langle \text{symbol} \rangle (\{ \langle \text{term} \rangle , \}^* \langle \text{term} \rangle) && \text{(Predicate Application)} \\ &| \langle \text{term} \rangle = \langle \text{term} \rangle && \text{(Set Element Equality)} \\ &| \langle \text{symbol} \rangle && \text{(Boolean Variable or Constant)} \end{aligned}$$

Predicate and function arguments are chosen among the set of all possible terms. A term is either a function symbol applied over a list of arguments or a variable/constant symbol:

$$\begin{aligned} \langle \text{term} \rangle &::= \langle \text{symbol} \rangle (\{ \langle \text{term} \rangle , \}^* \langle \text{term} \rangle) && \text{(Function Application)} \\ &| \langle \text{symbol} \rangle && \text{(Variable or Constant)} \end{aligned}$$

Finally, it should be noted that each of the *occurrences of a variable* appearing in the formulas of a theory can be either *bound* or *free*. An occurrence of a variable x is bound if this occurrence appears in a formula ϕ quantified by x , i.e. in a formula of the form $\forall x, \phi$ or $\exists x, \phi$. Otherwise, the occurrence is said to be free. For example, all the four variable occurrences appearing in the theory represented after this paragraph have been underlined. One can find one occurrence of the variable sp bound by the quantifier $\exists$, two occurrences of the variable p , the first one being free and the second one bound by the quantifier $\forall$, and one free occurrence of the variable s :

PREDICATE C **ON** $P \times S$;
VARIABLE sp **ON** SP ;
VARIABLE p **ON** P ;
VARIABLE s **ON** S ;
FORMULA $(\exists sp, (\underline{p} = \underline{sp})) \wedge (\forall p, C(\underline{p}, \underline{s}))$;

B.2 Semantics: Model, Satisfiability and Validity In order to reason using a given theory, one should provide a semantic for the symbols declared in the theory. This is done by defining a *model* for the theory, which associates each declared symbol with a corresponding set, constant, function or predicate value. A formula is *satisfied* by a model if, for any possible assignment of the free variable occurrences in the formula, the formula is true with respect to the semantic provided by the model. A logic is satisfied by a model if all its formulas are satisfied by the model. A formula is said to be *valid* if it is satisfied by any possible model. A logic is valid if all its formulas are valid.

For example, the theory that follows is satisfied by model M where symbol A represents the set of all the people who ever lived on Earth, the symbol C represents the set of all the book copies in the bookshop and the symbol W the predicate that returns true for each tuple (a, c) where a has written c :

PREDICATE W ON $A \times C$;
VARIABLE a ON A;
VARIABLE c ON C;
FORMULA $\exists a, W(a, c)$;

Indeed, for any value of the free variable c , i.e. for any book copy in the bookshop, there exists at least someone who has written this book. The formula $\exists a, W(a, c)$ is then satisfied by the model M, and as it is the only formula in the theory, the theory is also satisfied by this model. However, the formula $\exists a, W(a, c)$ is not valid as there exists at least one model that does not satisfy it. For example, the model N, where the symbols A and C represent both the set of all the people who ever lived on Earth and the symbol W the predicate that returns true for each tuple (a, c) where a is a sibling of c , does not satisfy the formula. Indeed, if c is chosen as someone who is an only child, the formula will be false.

C Describing Relational Operations using First-Order Logic

We now illustrate how each relational operator of the relational algebra can be described by an equivalent formula in first-order logic. In order to do so, we build a theory and a corresponding model that describes the relational operations illustrated at page 17 over the relational variables **shelf** and **copy**. We start by defining two predicate symbols SHELF0 and COPY0 and decide to consider only models where these two symbols represent respectively the characteristic functions of the relations **Shelf** and **Copy** that constituted the initial content of the variables **shelf** and **copy** in the example of page 17:

PREDICATE SHELF0 ON $I \times L$;
PREDICATE COPY0 ON $B \times T \times A \times I$;

Then, we add to the theory a predicate symbol LABELKING0, a set of intermediate predicate symbols LABELKING0a, LABELKING0b and LABELKING0c, as well as formulas linking each of these predicates together. The resulting addition to the theory can be found below this paragraph. The added formulas are crafted in a way such that the only model (among those where the symbols SHELF0 and COPY0 represent the **Shelf** and **Copy** relations) satisfying the augmented logic will be the one where the LABELKING0a symbol represents the characteristic function of the **Copy** $\times$ **Shelf** relation, the LABELKING0b symbol represents the characteristic function of the $\sigma_{P_{King}}(\text{Copy} \times \text{Shelf})$ relation, the LABELKING0c symbol represents the characteristic function of the $\sigma_{P_{SameIdentifier}}(\sigma_{P_{King}}(\text{Copy} \times \text{Shelf}))$ relation and finally the LABELKING0 symbol represents the characteristic function of the $\pi_{\text{Labels}}(\sigma_{P_{SameIdentifier}}(\sigma_{P_{King}}(\text{Copy} \times \text{Shelf})))$ relation.

Concretely, we enforce that this model is the only one that satisfies the logic by crafting the formulas so that they are equivalent to the predicative definition of the operators of the relational algebra given at subsection 1.1.3.

PREDICATE LABELKING0 ON L
PREDICATE LABELKING0a ON $B \times T \times A \times I$;
VARIABLE b ON B ;
VARIABLE t ON T ;
VARIABLE a ON A ;
VARIABLE $i1$ ON I ;
VARIABLE $i2$ ON I ;
VARIABLE l ON L ;
FORMULA
 $\forall b, \forall t, \forall a, \forall i1, \forall i2, \forall l, (\text{LABELKING0a}[b, t, a, i1, i2, l]$
 $\iff (\text{COPY0}[b, t, a, i1] \wedge \text{SHELF0}[i2, l]))$
PREDICATE LABELKING0b ON $B \times T \times A \times I \times I \times L$
VARIABLE i ON I ;
CONSTANT "Stephen King" ON A ;
FORMULA
 $\forall b, \forall t, \forall a, \forall i, (\text{LABELKING0b}[b, t, a, i]$
 $\iff (\text{LABELKING0a}[b, t, a, i] \wedge (a = \text{"Stephen King"})))$
PREDICATE LABELKING0c ON $B \times T \times A \times I \times I \times L$
FORMULA
 $\forall b, \forall t, \forall a, \forall i1, \forall i2, \forall l,$
 $(\text{LABELKING0c}[b, t, a, i1, i2, l] \iff (\text{LABELKING0b}[b, t, a, i1, i2, l] \wedge (i1 = i2)))$
FORMULA
 $\forall b, \forall t, \forall a, \forall i1, \forall i2, \forall l, (\text{LABELKING0}[l] \iff \text{LABELKING0c}[b, t, a, i1, i2, l])$

Similarly, we can write formulas constraining the predicate symbols COPY1 and COPY2 to represent the content of the `copy` variable respectively after the Stephen King's book being bought and after the new book copy to sell being received by the bookshop:

PREDICATE COPY1a ON $B \times T \times A \times I$;
CONSTANT 87567 ON N ;
FORMULA
 $\forall b, \forall t, \forall a, \forall i, (\text{COPY1a}[b, t, a, i] \iff (\text{COPY0}[b, t, a, i] \wedge (b = 87567)))$
PREDICATE COPY1 ON $B \times T \times A \times I$;
FORMULA
 $\forall b, \forall t, \forall a, \forall i, (\text{COPY1}[b, t, a, i] \iff (\text{COPY0}[b, t, a, i] \wedge (\neg \text{COPY1a}[b, t, a, i])))$
PREDICATE COPY2 ON $B \times T \times A \times I$;
CONSTANT 98215 ON N ;
CONSTANT "A Game of Thrones" ON T ;
CONSTANT "George R. R. Martin" ON A ;
CONSTANT 13 ON I ;
FORMULA
 $\forall b, \forall t, \forall a, \forall i,$
 $(\text{COPY2}[b, t, a, i] \iff (\text{COPY1}[b, t, a, i] \vee (b = 98215 \wedge t = \text{"A Game of Thrones"} \wedge a = \text{"George R. R. Martin"} \wedge i = 13)))$

First-order logic can thus offer a declarative approach to perform operations over relations, by stating precisely, via a logical formula, how the result relation should be. In order to compute this result relation, one can use the corresponding operators in the relational algebra, which essentially offer an imperative procedure to do so.

1.1.5 Uniqueness and Referential Integrity

Let us consider the relational database we have built for the bookshop, composed of the relational variables `shelf` and `copy`, and let us suppose again that these variables contain the relations `Shelf` and `Copy`, representing the current resources available in the bookshop. Updating the content of these variables, using the relational algebra, can sometimes result in the relational variables to contain relations that do not correspond to a real state of the bookshop's resources. For example, let us consider the following operation:

$$\text{copy} = \text{copy} \cup \{(45536, \text{Foundation}, \text{Isaac Asimov}, 45)\}$$

that leads the `shelf` and `copy` variables to contain the following `Shelf` and `Copy` relations:

Relation Shelf	
Identifier	Label
13	Fantasy
14	Horror

Relation Copy			
Barcode	Title	Author	Identifier
45536	Harry Potter and the Deathly Hallows	J.K. Rowling	13
45536	Foundation	Isaac Asimov	45
45535	Harry Potter and the Deathly Hallows	J.K. Rowling	13
56563	The Return of the King	J.R.R. Tolkien	13
87567	The Shining	Stephen King	14

These relations do not represent a real state of the resources available in the bookshop. First, two book copies share the same barcode (45536), whereas the barcodes used in the bookshop are unique. Secondly, the copy of the Asimov's book is supposed to be stored in a shelf with identifier 45, whereas no shelf with identifier 45 exists in the *Shelf* relation. When the domain of interest prevents the tuples stored in a relational variable from sharing a single value for one or a set of attributes, the relation is said to be subject to a so-called *uniqueness* constraint. Any relation that is assigned to a relational variable must obey all the uniqueness integrity constraints existing for this variable. Uniqueness constraints can be easily expressed using first-order logic. For example, for the `copy` variable and the uniqueness constraint stating that barcodes should be unique, we have the following theory, with the usual symbol semantics, and where the `COPY` predicate is the characteristic function of a relation to be stored in the `copy` variable:

PREDICATE COPY ON $B \times T \times A \times I$;

VARIABLE b1 ON B; **VARIABLE** b2 ON B;

VARIABLE t1 ON T; **VARIABLE** t2 ON T;

VARIABLE a1 ON A; **VARIABLE** a2 ON A;

VARIABLE i1 ON I; **VARIABLE** i2 ON I;

FORMULA

$\forall b1, \forall b2, \forall t1, \forall t2, \forall a1, \forall a2, \forall i1, \forall i2,$

$(\text{COPY}[b1, t1, a1, i1] \wedge \text{COPY}[b2, t2, a2, i2] \wedge (b1 = b2)) \implies ((t1 = t2) \wedge (a1 = a2) \wedge (i1 = i2))$

When the domain of interest constrains each of the tuples stored in a relational variable to reference another tuple in another relational variable, the first relational variable is said to be subject to a *referential integrity* constraint. Any relation that is assigned to

a relational variable must obey all the referential integrity constraints existing for this variable. Referential integrity constraints can be easily expressed using first-order logic. For example, for the **shelf** and **copy** variables and the referential integrity constraint stating that each book copy should reference one shelf, we have the following theory, with the usual symbol semantics, and where the **SHELF** and **COPY** predicates are respectively the characteristic functions of relations to be stored in the **shelf** and **copy** variables:

PREDICATE SHELF ON $I \times L$;
PREDICATE COPY ON $B \times T \times A \times I$;
VARIABLE b ON B;
VARIABLE t ON T;
VARIABLE a ON A;
VARIABLE i ON I;
VARIABLE l ON L;
FORMULA
 $\forall b, \forall t, \forall a, \forall i, \text{COPY}(b, t, a, i) \implies (\exists l, \text{SHELF}(i, l))$

In practice, many different other kinds of integrity constraints can be defined over relational variables to prevent the representation of invalid information. In a relational database, any modification of the content of a relational variable that violates one of the integrity constraints collected for the variable is forbidden.

For a given relational variable, the exact type of this variable (i.e. the ordered list of sets from which are built the relations that can be stored in the variable) and the integrity constraints defined for this variable constitute the so-called *schema* of the relational variable. The set of relational variables defined in a relational database and their associate schema constitute the schema of the database. The relations stored in the variables must be compliant with the database schema and constitute the content of the database, manipulated using either the relational algebra or calculus.

1.2 Relational DBMS and SQL

A database can be described in practice as an organized repository that makes it possible to store data representing information about large quantities of inter-related facts. The stored information must typically be permanently available for retrieval to a large number of applications accessing the database. Some of these applications are also responsible for modifying the content of the database, adding, updating or deleting information either to represent newly occurring facts or to remove outdated facts.

In the previous section, we have studied a theoretical design pattern to build a database, called the relational model. A database designed according to this model is called a relational database and is composed of a set of relational variables containing the stored information, which can be queried and modified, according to the database schema, using the relational algebra and the relational calculus. The modeling approach proposed by the relational model (where the domain of interest is represented by a set of relational variables and their associate schema) and the languages offered by the relational model to manipulate the stored information (the relational algebra and calculus) are universal tools to build a database, as they can be applied whatever the information stored in the built database are about.

Since its introduction by Codd in the 1970s, the relational model has become the de-facto standard for practical database design. In this section, we study how a relational database can be built in practice. In subsection 1.2.1, we present briefly *relational database management systems*, i.e. very popular software utilities used by developers to create and manage fully-functional relational databases. These software utilities are typically based on a standardized language for creating and managing databases, called SQL. We study the SQL language in subsection 1.2.2. Finally, we detail in subsection 1.2.3 how applications can interact smoothly with a relational database management system, using interfaces like *embedded SQL* or *CLI* and mechanisms like *ACID transactions*.

1.2.1 Relational Database Management Systems

A database management system (DBMS) is a standard software utility that enables one to create and manage a database, whatever the information to be stored in the database is about. It can be used to create the data structures composing the database and it offers a standardized interface for applications to access and modify the content of these structures in an organized and controlled way. In concrete terms, a database management system substantially offers:

- A *data description language* (DDL), offering primitives to define a set of standardized data structures, implementing a given *data model*, i.e. an universal abstraction for representing information, whatever this information is about.
- A *data modification language* (DML), offering standard primitives to retrieve, add, update and delete data stored in the defined data structures.
- Technical mechanisms ensuring data storage, DDL and DML statement execution and controlled access to a reliable database (e.g. access APIs, concurrent access management, access right management, crash recovery, etc.).

A relational database management system is a DBMS that makes use of the relational modeling approach described by the relational model as a data model, of relational variables

and their associate schema as a DDL and of a language based on the relational algebra or calculus as a DML. Since the birth of the relational model in the 1970's, relational DBMSs have become the almost monopolistic leaders of the market of DBMSs. The reasons advocated for explaining this success are often the simple, universal, rigorous and high-level paradigm that the relational model offers to create, access and update databases. At the kernel of all the current relational DBMSs available on the market stands a standardized DDL and DML, implementing the principles of the relational model, called the Structured Query Language (SQL). SQL was born in the 1970s from research to provide an english-based syntax for the relational algebra and calculus, which had previously been introduced by Codd. As SQL was deployed in many of the relational DBMSs appearing at that time, the language emerged as a standard, and was finally normalized by ISO in 1987. The standard has received seven revisions since its introduction, the latest in 2011, and is available from ISO [ISO, 2011].

The DDL and DML offered by SQL are based on the relational model, namely integrating the ability to define relational variables and their associate schema (SQL DDL) as well as to query and update the content of these variables using primitives founded on the relational algebra and calculus (SQL DML). Nevertheless, SQL DDL and DML have also been enriched with several non-relational features [Connolly and Begg, 2005], like missing information management, sorting or grouping abilities, turing-complete capabilities etc. Moreover, several non-relational DBMSs have appeared on the market during the last decades, leading relational DBMSs manufacturers to integrate some of the features proposed by these concurrent tools into their own products. As a consequence, the ISO SQL DDL and DML have also been added elements extracted from non-relational DBMSs [Hainaut, 2012], like object-oriented extensions, recursive queries or XML tools. Finally, primitives to handle the additional technical mechanisms typically offered by a DBMS, notably to ensure a controlled access to a reliable database, have also been added to the ISO SQL language [Connolly and Begg, 2005], in order to standardize these primitives between different relational DBMS manufacturers.

Compared to the initial relational model, SQL is thus a much more powerful but also a notoriously complex, ambiguous and redundant language. Moreover, it is typically not portable between different relational DBMS manufacturers, as each of them usually offers its own SQL *dialect*, which only partially obeys the ISO standard and contains proprietary extensions. In the next subsection, following [Connolly and Begg, 2005], we provide an intuitive description of the core DDL and DML parts of the ISO SQL, which are implemented quite homogeneously by all the DBMS manufacturers, and emphasize the similarities and differences existing with the relational model.

1.2.2 The SQL Language

A SQL DDL

The SQL Data Description Language essentially offers a syntax to create relational variables and their associated schemas. A relational variable containing a relation (i.e. a set of tuples) is represented in SQL by a *table* containing a set of *rows*. The *columns* of the table correspond to the attributes of the relations stored in the variable. Uniqueness constraints over some attributes are expressed using so-called *primary key* or *unique* constraints over the corresponding columns. Referential integrity constraints are expressed similarly using so-called *foreign key* constraints. Other integrity constraints can be expressed using notably *check* constraints. For example, we defined in the previous section a database composed

of two relational variables **shelf** and **copy** to represent the state of the bookshop. The schema of this relational database was:

Variable **shelf** $\in$ **Identifiers** $\times$ **Labels**

Uniqueness: Shelves standing in the bookshop have unique identifiers.

Variable **copy** $\in$ **Barcodes** $\times$ **Titles** $\times$ **Authors** $\times$ **Identifiers**

Uniqueness: Book copies available in the bookshop have unique barcodes.

Referential Integrity:

A book copy's identifier references a shelf stored in the **shelf** variable.

In SQL, the data structures for two corresponding shelf and copy tables can be created using the two following CREATE TABLE statements:

```
CREATE TABLE shelf (
  identifier INTEGER NOT NULL,
  label NVARCHAR2(50) NOT NULL,
PRIMARY KEY (identifier),
CHECK(identifier >= 0));
```

```
CREATE TABLE copy (
  barcode INTEGER NOT NULL,
  title NVARCHAR2(250) NOT NULL,
  author NVARCHAR2(100) NOT NULL,
  identifier INTEGER NOT NULL,
PRIMARY KEY (barcode),
FOREIGN KEY (identifier) REFERENCES shelf);
```

Each column in a table is associated with a definite datatype. For example, in the book table, titles are represented by character strings of maximum 250 characters (**NVARCHAR2(250)**). Similarly, identifiers in the shelf table are represented by integers (**INTEGER**), but as shelves are never identified by negative numbers in the bookshop, a so-called *domain integrity constraint* is added, via a check constraint, to prevent the existence of rows with a negative value of identifier.

The **CREATE TABLE** statement is the main statement offered by SQL DDL. The language also offers other common statements that we discuss briefly. The **ALTER TABLE** and **DROP TABLE** statements enable respectively modifying a table definition and removing a table definition from the database schema. The **CREATE SCHEMA** and **DROP SCHEMA** statements enable creating and deleting the schema of the database. The **CREATE VIEW** and **DROP VIEW** statements enable one to create and delete views, i.e. mock tables gathering data from other tables, which can be used as real tables for many operations. Finally, the **CREATE DOMAIN**, **ALTER DOMAIN** and **DROP DOMAIN** statements make it possible to create, modify and delete new custom datatypes for table columns. In the remainder of this section, we provide a detailed syntax for the **CREATE TABLE** statement, using a BNF grammar, and discuss the main differences with the relational algebra.

The created table is given a name and associated with a set of columns and integrity constraint definitions:

$\langle \text{create-table} \rangle ::= \text{CREATE TABLE } \langle \text{table-name} \rangle (\{ \langle \text{column} \rangle , \}^* \langle \text{column} \rangle \langle \text{constraints} \rangle) ;$

Each defined column is given a name and a type:

$\langle column \rangle ::= \langle column-name \rangle \langle type \rangle \{ \text{NOT NULL} \}^? \langle default \rangle^? ,$

The main general types proposed in the ISO standard are

$\langle type \rangle ::= \langle boolean \rangle$
 $\quad | \langle character \rangle$
 $\quad | \langle bit-vector \rangle$
 $\quad | \langle number \rangle$
 $\quad | \langle point-in-time \rangle$
 $\quad | \langle interval-of-time \rangle$
 $\quad | \langle large-object \rangle$

DBMS manufacturers typically offer many concrete variants of these general types. The large object type enables storing content of potentially very large size, like textual or multimedia content. The NOT NULL keyword makes it possible to forbid the use of the *null* value for a given column in the rows stored in the table. The *null* value is a generic value to be used as a placeholder in case of a missing datum. For example, if the label column in the *shelf* table was defined as

label **NVARCHAR2**(50),

this would allow the table to contain rows that represent shelves whose label information is missing, like (15,*null*). The ability to use *null* values is one of the important features that can be used in SQL and that are not part of the standard relational model. DBMSs also typically enable one to define a default column value, to be used when one wants to insert a row in the table without providing a value for this column:

$\langle default \rangle ::= \text{DEFAULT } \langle value \rangle$

For example, if the *label* column in the *shelf* table was defined as

label **NVARCHAR2**(50) **NOT NULL DEFAULT** 'Other Topics',

asking the DBMS to insert a row (16,?) in the shelf table would result in the (16,'Other Topics') to be actually inserted in the table.

The integrity constraints associated to a table can be of four different types: primary key, unique, foreign key or check constraints:

$\langle constraints \rangle ::= \langle primary-key \rangle^? \langle unique \rangle^* \langle foreign-key \rangle^* \langle check \rangle^*$

An uniqueness constraint over one or several columns can be declared using either a **PRIMARY KEY** or **UNIQUE** constraint:

$\langle primary-key \rangle ::= , \text{PRIMARY KEY } (\langle column-name-list \rangle)$

$\langle unique \rangle ::= , \text{UNIQUE } (\langle column-name-list \rangle)$

The behavior of these two kinds of constraints is essentially similar, but SQL allows to declare at most one primary key per table, supposed to be used for defining the most representative set of columns that enables one to identify semantically the objects represented by the rows in the table. There is no limit over the number of unique constraints that can be defined for a table. When no primary key and no unique constraint is defined for a table, the table is allowed to contain duplicate rows. This is another important difference between SQL and the relational model, where relations are sets of distinct tuples. Referential integrity constraints over one or several columns can be declared using a **FOREIGN KEY** constraint:

$$\langle \textit{foreign-key} \rangle ::= , \textbf{FOREIGN KEY} (\langle \textit{column-name-list} \rangle) \\ \textbf{REFERENCES} \langle \textit{table-name} \rangle \{ (\langle \textit{column-name-list} \rangle) \}^? \\ \langle \textit{on-update} \rangle^? \langle \textit{on-delete} \rangle^?$$

Multi-column foreign keys can be declared, like in

FOREIGN KEY (author,identifier) **REFERENCES** shelf(label,identifier) ,

which would prevent a row (23456,‘The Shining’,‘Stephen King’,15) to be present in the copy table if the row (15,‘Stephen King’) is not present in the shelf table. The referential behavior associated to a foreign key (at choice **CASCADE**, **SET NULL**, **SET DEFAULT** or **NO ACTION**) indicates how the DBMS must react when a row referenced by other ones is deleted or sees its referenced column value(s) updated. This behavior, which can be different from a foreign key to another and between update and delete operations, can be specified using the following rules:

$$\langle \textit{on-update} \rangle ::= \textbf{ON UPDATE} \langle \textit{rule} \rangle$$

$$\langle \textit{on-delete} \rangle ::= \textbf{ON DELETE} \langle \textit{rule} \rangle$$

$$\langle \textit{rule} \rangle ::= \textbf{CASCADE} \\ | \textbf{SET NULL} \\ | \textbf{SET DEFAULT} \\ | \textbf{NO ACTION}$$

leading, for example, to a foreign key definition similar to

FOREIGN KEY (identifier) **REFERENCES** shelf **ON DELETE CASCADE**,

The **NO ACTION** behavior is the default setting, where the DBMS rejects any update/delete operation over a referenced row, as in the relational model. The other behaviors allow the update/delete operation to be executed. The **CASCADE** behavior triggers the necessary update/deletion actions over the referencing rows to reflect the new content of the referenced table, letting the database in a consistent state. The **SET DEFAULT** and **SET NULL** behaviors provoke the referencing column values to be set respectively to their default value or to the *null* value, for each row referencing an updated/deleted row. Other integrity constraints can be defined using the following syntax:

$$\langle \textit{check} \rangle ::= , \textbf{CHECK} (\langle \textit{condition} \rangle)$$

Any operation over the table’s content that makes the condition false will be rejected. It should be noted that the ISO SQL standard provides more advanced mechanisms, like triggers, to express integrity constraints.

B SQL DML

The SQL Data Modification Language offers a syntax based on the relational algebra and calculus to query and update the data stored in the tables created using SQL DDL. This syntax includes **SELECT** statements for data querying and **INSERT**, **UPDATE** and **DELETE** statements for data updating. We detail these statements and emphasize the differences with the relational algebra and calculus in this subsection.

B.1 SQL Queries A **SELECT** query is a DML statement that enables one to retrieve data from one or several tables. The result data are organized and displayed as if they composed the content of an hypothetical single output table. **SELECT** queries essentially offer syntactic sugar for the relational algebra. For example, in the previous section (page 17), we have used the relational algebra's operators over the **shelf** and **copy** relational variables to establish the label of the shelves in which the Stephen King's books can be found in the bookshop. The resulting relation was computed into the temporary relational variable **labelKing** as:

$$\text{labelKing} = \pi_{\text{Labels}}(\sigma_{P_{\text{SameIdentifier}}}(\sigma_{P_{\text{King}}}(\text{copy} \times \text{shelf})))$$

The SQL table corresponding to the relation stored in the **labelKing** variable can be obtained using the following **SELECT** query, composed of three successive *clauses* (the **SELECT** clause, the **FROM** clause and the **WHERE** clause):

SELECT DISTINCT label

FROM shelf, copy

WHERE shelf.identifier = copy.identifier **AND** copy.author = 'Stephen King'

The **FROM** clause defines the table from which the data should be retrieved. When several input tables are indicated after the **FROM** keyword, like here with the shelf and book tables, the data should then be retrieved from an hypothetical table containing the product of the input tables. The **FROM shelf, copy** clause is thus equivalent to the relational operation **copy** $\times$ **shelf**. The **WHERE** clause selects the rows in the table defined by the **FROM** clause that make the predicate specified after the **WHERE** keyword true. The **FROM shelf, copy** clause with the **WHERE shelf.identifier=copy.identifier AND copy.author='Stephen King'** clause are thus equivalent to the relational operation $\sigma_{P_{\text{SameIdentifier}}}(\sigma_{P_{\text{King}}}(\text{copy} \times \text{shelf}))$. Finally, the **SELECT** clause returns the table containing the rows selected by the **FROM** and **WHERE** clauses restricted to the columns specified after the **SELECT DISTINCT** keyword. The whole **SELECT** query specified above is thus equivalent to the relational operation $\pi_{\text{Labels}}(\sigma_{P_{\text{SameIdentifier}}}(\sigma_{P_{\text{King}}}(\text{copy} \times \text{shelf})))$.

If the product, selection and projection operators of the relational algebra can be expressed by a $\langle \text{select-from-where} \rangle$ block, the two remaining relational operators (set union and difference) can be used to combine $\langle \text{select-from-where} \rangle$ blocks:

$$\begin{aligned} \langle \text{select-combination} \rangle ::= & \langle \text{select-combination} \rangle \text{ UNION } \langle \text{select-combination} \rangle \\ & | \langle \text{select-combination} \rangle \text{ EXCEPT } \langle \text{select-combination} \rangle \\ & | \langle \text{select-from-where} \rangle \end{aligned}$$

For example, the following query:

```
(SELECT DISTINCT identifier
FROM copy
WHERE author = 'Stephen King')
UNION
(SELECT DISTINCT identifier
FROM copy
WHERE author = 'J.R.R. Tolkien')
```

is equivalent to the following relational operation:

$$\pi_{\text{Identifier}}(\sigma_{P_{\text{King}}}(\text{copy})) \cup \pi_{\text{Identifier}}(\sigma_{P_{\text{Tolkien}}}(\text{copy}))$$

SQL queries constitute a relational complete language. A more complete syntax for queries is given by:

$\langle query \rangle ::= \langle select-combination \rangle \langle order-by \rangle$

$\langle select-from-where \rangle ::= \langle select \rangle \langle from \rangle \langle where \rangle \langle group-by \rangle \langle having \rangle$

$\langle select \rangle ::= \text{SELECT DISTINCT? } \langle expression-list \rangle$

$\langle from \rangle ::= \text{FROM } \langle table-name-list \rangle$

$\langle where \rangle ::= \text{WHERE } \langle predicate \rangle$

$\langle group-by \rangle ::= \text{GROUP BY } \langle column-name-list \rangle$

$\langle having \rangle ::= \text{HAVING } \langle predicate \rangle$

$\langle order-by \rangle ::= \text{ORDER BY } \langle column-name-list \rangle$

We illustrate the main additional possibilities offered by this syntax, compared to the relational algebra, by demonstrating the execution of a set of representative sample SQL queries over the following two tables:

Table <i>shelf</i>	
identifier	label
13	Fantasy
14	Horror
15	<i>null</i>

Table <i>copy</i>			
barcode	title	author	identifier
45536	Harry Potter and the Deathly Hallows	J.K. Rowling	13
45535	Harry Potter and the Half-Blood Prince	J.K. Rowling	15
56563	The Return of the King	J.R.R. Tolkien	13
56564	The Two Towers	J.R.R. Tolkien	13
56565	The Fellowship of the Ring	J.R.R. Tolkien	13
87567	The Shining	Stephen King	14

A select query basically returns a list of rows composing a hypothetical result table. Unlike the relational algebra (acting on relations defined as sets of distinct tuples), the returned rows can contain duplicates. For example, the following query

SELECT author
FROM copy

will return the table

author
J.K. Rowling
J.K. Rowling
J.R.R. Tolkien
J.R.R. Tolkien
J.R.R. Tolkien
Stephen King

The **DISTINCT** keyword enables recovering the relational algebra's behavior and the query

```
SELECT DISTINCT author
FROM copy
```

returns

author
J.K. Rowling
J.R.R. Tolkien
Stephen King

Compared to the relational algebra, SQL also adds the ability to use DBMS built-in functions or predicates in selection and projection. For example the following query:

```
SELECT DISTINCT author,CHARACTER_LENGTH(author)
FROM copy
WHERE CHARACTER_LENGTH(author) > 12
```

uses the **CHARACTER_LENGTH** function, which computes the length of the provided character string argument, as well as the well-known > predicate and returns the following table:

author	CHARACTER_LENGTH (author)
J.R.R. Tolkien	14

Among the functions typically employed in SQL queries, *aggregate functions* can be used to compute statistics over the rows returned by the query. For example, the **COUNT**(*) function computes the number of rows in the table returned by the query. *Subqueries* are SQL queries used as functions in other queries. The following query computes the number of book copies written by J.K. Rowling that are stored in shelves labeled as “Fantasy” and it makes use of an aggregate function and of a subquery:

```
SELECT COUNT(*)
FROM copy
WHERE author='J.K. Rowling' AND identifier IN (SELECT identifier
                                                FROM shelf
                                                WHERE label='Fantasy')
```

The returned table is:

COUNT (*)
1

The possible presence of *null* values in SQL tables makes predicate evaluation more complex than in the relational algebra. For example, the following query:

```
SELECT identifier,label
FROM shelf
WHERE NOT (label = 'Fantasy')
```

returns the table

identifier	label
14	Horror

which does not contain the row “(15,*null*)”. Indeed, the truth value of

NOT (**null** = 'Fantasy')

is unknown, as *null* corresponds to a missing datum, leading to rejection of the row “(15,*null*)” by the **WHERE** clause. The **IS NULL** predicate is used to test if the provided argument is valued to *null* or not. For example:

```
SELECT identifier,label
FROM shelf
WHERE (label IS NULL) OR (NOT (label = 'Fantasy'))
```

returns

identifier	label
14	Horror
15	<i>null</i>

Compared to the relational algebra, SQL also introduces the ability to group rows, i.e. to merge several rows sharing the same values for one or several given columns, using the **GROUP BY** keyword, followed by the column(s) to be considered for merging. The grouped rows form new rows that can be filtered using a **HAVING** keyword, followed by a predicate to be verified by the filtered rows. Aggregate functions can be applied to compute statistics within each group of merged rows. For example, the following query returns the list of all authors for which there are at least two book copies in the bookshop, as well as the total number of books present for these authors:

```
SELECT author,COUNT(barcode)
FROM copy
GROUP BY author
HAVING COUNT(barcode) > 1
```

which returns:

author	COUNT(barcode)
J.K Rowling	2
J.R.R. Tolkien	3

By default and according to the relational algebra, the rows returned by an SQL query are not arranged in any particular order. Nevertheless, an **ORDER BY** clause can be used to specify columns on which the returned rows must be sorted. For example, the following query returns the list of different titles available in the bookshop, sorted by alphabetical order:

```
SELECT DISTINCT title
FROM copy
ORDER BY title
```

which returns

title
Harry Potter and the Deathly Hallows
Harry Potter and the Half-Blood Prince
The Fellowship of the Ring
The Return of the King
The Shining
The Two Towers

B.2 SQL INSERT, UPDATE and DELETE statements In the relational model, the content of a relational variable can be updated with a new relation to reflect the changing state of the domain of interest. The new relation can be computed from the previous relation in the variable using the relational algebra. In SQL DML, three statements can be used to modify the content of a table: an **INSERT** statement makes it possible to add new rows to a given table, an **UPDATE** statement makes it possible to update the column values of the rows in a given table and a **DELETE** statement makes it possible to delete rows from a given table. The relational operation used in the previous section when updating the **copy** relational variable to reflect buying the book copy with barcode 87567

$$\text{copy} = \text{copy} - (\sigma_{P_{87567}}(\text{copy}));$$

is equivalent to the following **DELETE** statement over the **copy** table, which deletes any row in the table with a barcode value of 87567:

DELETE FROM **copy**
WHERE barcode = 87567

Similarly, the other relational operation used in the previous section

$$\text{copy} = \text{copy} \cup \{(98215, \text{A Game of Thrones}, \text{George R. R. Martin}, 13)\}$$

adding a tuple to reflect the arrival of a new book copy in shelf 13 is equivalent to the following **INSERT** statement, which adds the corresponding row in the table **copy**:

INSERT INTO **copy**
VALUES (98215, 'A Game of Thrones', 'George R. R. Martin', 13)

Conversely, the following **UPDATE** statement

UPDATE **copy**
SET identifier = 14
WHERE author = 'J.K. Rowling'

which sets the value of the column identifier to 14 for all rows where author is J.K. Rowling, is equivalent to the following relational operations over the **copy** variable:

$$\text{copy} = \text{copy} - \sigma_{P_{\text{Rowling}}}(\text{copy}) \cup (\pi_{\text{Barcodes, Titles, Authors}}(\sigma_{P_{\text{Rowling}}}(\text{copy})) \times \{(14)\})$$

The detailed syntax for **INSERT**, **UPDATE** and **DELETE** statements, given by the following BNF, is very simple and almost entirely encompassed by the three examples that we have just given:

$\langle \text{insert-update-delete} \rangle ::= \langle \text{insert} \rangle$
 $\quad | \quad \langle \text{update} \rangle$
 $\quad | \quad \langle \text{delete} \rangle$

$\langle \text{insert} \rangle ::= \text{INSERT INTO } \langle \text{table-name} \rangle \{(\{ \langle \text{column-name} \rangle, \}^* \langle \text{column-name} \rangle)\}^? \langle \text{data} \rangle$

$\langle \text{data} \rangle ::= \text{VALUES } (\langle \text{value-list} \rangle)$
 $\quad | \quad \langle \text{query} \rangle$

$\langle \text{update} \rangle ::= \text{UPDATE } \langle \text{table-name} \rangle$
 $\quad \text{SET } \{ \langle \text{column-name} \rangle = \langle \text{expression} \rangle , \}^* \langle \text{column-name} \rangle = \langle \text{expression} \rangle$
 $\quad \langle \text{where} \rangle^?$

$\langle delete \rangle ::= \text{DELETE FROM } \langle table-name \rangle \langle where \rangle?$

One should note the ability to use the rows returned by an SQL **SELECT** query as input for an **INSERT** statement like in

```
INSERT INTO author
SELECT author,COUNT(barcode)
FROM copy
GROUP BY author
```

(supposing the existence of an author table)

and the possibility to use DBMS functions in the SET clause of **UPDATE** statements, like in:

```
UPDATE copy
SET identifier = identifier + CHARACTER_LENGTH(author)
```

Finally, one should note that any **INSERT**, **UPDATE** and **DELETE** statement that makes the resulting table content violate an integrity constraint defined in the DDL schema is typically rejected by the DMBS. For example, considering the following table definition:

```
CREATE TABLE shelf (
  identifier INTEGER NOT NULL,
  label NVARCHAR2(50) NOT NULL,
PRIMARY KEY (identifier),
CHECK(identifier  $\geq$  0));
```

and the following table content:

Table <i>shelf</i>	
identifier	label
13	Fantasy
14	Horror

the following statement will be rejected:

```
INSERT INTO shelf
VALUES (13,'Tetrapilectomy')
```

as it violates the **PRIMARY KEY** constraint, as well as the statement

```
UPDATE shelf
SET identifier = -5
```

that violates the **CHECK** constraint. The DBMS behavior for **FOREIGN KEY** constraints depends on the referential behavior defined in the DDL schema. For example, considering the following table definition, with a 'CASCADE' referential integrity behavior for 'DELETE' statements:

```
CREATE TABLE copy (
  barcode INTEGER NOT NULL,
  title NVARCHAR2(250) NOT NULL,
  author NVARCHAR2(100) NOT NULL,
  identifier INTEGER NOT NULL,
PRIMARY KEY (barcode),
FOREIGN KEY (identifier) REFERENCES shelf ON DELETE CASCADE);
```

and the following table content:

Table <i>copy</i>			
barcode	title	author	identifier
45536	Harry Potter and the Deathly Hallows	J.K. Rowling	13
56563	The Return of the King	J.R.R. Tolkien	13
56564	The Two Towers	J.R.R. Tolkien	13
56565	The Fellowship of the Ring	J.R.R. Tolkien	13
87567	The Shining	Stephen King	14

the following statement:

DELETE FROM shelf
WHERE identifier = 13

will be successfully executed and lead to the following table content:

Table <i>shelf</i>	
identifier	label
14	Horror

Table <i>copy</i>			
barcode	title	author	identifier
87567	The Shining	Stephen King	14

1.2.3 SQL in Software Applications

A Using SQL Code within Application Code

If relational DBMSs make it possible to create, manage and access relational databases, these actions are often triggered by external software applications. These applications must be written in a high-level programming language, which usually does not provide native primitives to make distant SQL calls to the DBMS. Integrating SQL code with application code requires thus to extend the programming power of the application language with such primitives. In particular, this integration should enable applications to submit SQL code that depends on the content of the application's variables, and, conversely, it should be possible for applications to copy into their variables the result obtained for a submitted SQL statement (the rows returned by an SQL query or a pass/fail message for the other SQL operations).

Many utilities have been developed in practice to bridge the gap between the many existing programming languages and DBMSs. Conceptually, the provided ways of executing an SQL statement from an application can be classified according to two dimensions: static vs dynamic SQL (depending on the way the SQL code is built within the application) and call level interface (CLI) vs embedded SQL (depending on the way the built SQL code is handled to be transmitted to the DBMS):

- In **static SQL**, the application programmer writes the SQL code to be executed when writing the application's code. The value of the constants used in the SQL clauses can be defined parametrically, at programming time, as a function of the application's variables. The SQL code is compiled together with the application and the compiled SQL statement is stored by the DBMS in the database as metadata. During the application execution, the DBMS is asked by the application to execute the stored compiled code and eventually provided with concrete values for the constants defined

parametrically. In **dynamic SQL**, the application builds dynamically a character string containing the SQL statement code, and asks the DBMS to compile and execute it at runtime.

- In **embedded SQL**, the SQL statements appear in the application as if they were native code. An application with embedded code must first be processed by an ad-hoc *pre-compiler*, to replace the SQL code by native code, and then compiled and executed normally. Embedded SQL is compatible both with static and dynamic SQL. In **CLI SQL**, the SQL statements appear as parameters of functions executed at runtime, provided by an existing *API* in native code. These functions can be called either to execute the SQL statement provided as a parameter or to retrieve the result of previously executed SQL code. CLI SQL is only compatible with dynamic SQL.

Dynamic SQL is more flexible and more general than static SQL. In practice, it facilitates the development of applications where the interaction pattern with the database is only known at runtime, like for example applications where users unaware of SQL can build and execute queries in a simplified way, using a graphical interface. On the other hand, dynamic SQL requires the SQL code to be compiled at each execution of the application, whereas it is compiled once for all the possible executions in static SQL. Moreover, dynamic SQL enables one to build the SQL code from any character string built in the applications's variables, which implies a less readable, harder to test and harder to maintain code. In addition, the application's inputs can be used as SQL code to be executed, opening the door for writing unsafe code, vulnerable to SQL injections attacks [Halfond et al., 2006]. The SQL ISO standard offers a specification to integrate SQL code in applications written in several languages like C or Java. This standard integration mode follows the embedded SQL paradigm and enables both static and dynamic SQL. The industry standards ODBC (see e.g. [Connolly and Begg, 2005]) and JDBC [Oracle, 2014] offer very popular frameworks for application-DBMS interaction, following the CLI SQL paradigm.

B Transaction Management Primitives

When an application requests the execution of an SQL statement to a relational DBMS, the DBMS offers a set of standardized consistency and reliability guarantees to the application's programmer, commonly known as the four ACID properties [Haerder and Reuter, 1983]. These properties are very common in relational DBMSs, so that they are often described as an irremovable complement to the relational model. The four ACID properties are:

Atomicity The SQL statement is an indivisible unit of execution. It is either executed totally or not executed at all, but never executed partially. It may not be executed at all if a failure arises during execution.

Consistency If the database is in a coherent state, then it will remain coherent after the execution of the SQL statement. The SQL statement is never executed if its execution would violate an integrity constraint defined in the database schema.

Isolation The SQL statement can be written considering the simplifying assumption that there is no concurrent applications acting over the database. Problems due to concurrent accesses and modifications of the database are silently solved by the DBMS.

Durability Once the SQL statement is successfully executed, the DBMS guarantees that the resulting effects will not be lost in case of failure.

Nevertheless, in some cases, implementing what is a single logical operation over the database may require the application to execute several successive SQL statements. A failure occurring during the execution of these statements (like one statement violating an uniqueness or referential integrity constraint) may let the database in a state where the logical operation has been only partially executed. Similarly, interferences with operations performed concurrently by other applications may lead to inconsistencies. To prevent such problems, DBMSs provide application programmers with the ability to define *transactions*, i.e. a group of one or more successive SQL statements that constitute a unitary logical operation for which the four ACID properties hold. The ISO SQL standard provides a standard API specification for defining transactions. This API specification is based on two standard primitives, *commit* and *rollback*, and the following transaction handling model:

1. The application begins its first transaction by requesting the execution of a (transaction-initiating) SQL statement.
2. A call to *commit* tells the DBMS to end the current transaction and to make permanent all the changes caused by the application to the database since the beginning of the transaction. A call to *rollback* backs out all the changes caused by the application to the database since the beginning of the transaction. The first (transaction-initiating) SQL statement emitted by the application after a call to *commit* or *rollback* starts a new transaction.
3. Normal termination of the application automatically commits the last transaction¹. Abnormal termination of the application automatically rolls back the last transaction.

More complex transaction models than the standard approach described here have been developed for advanced application needs (see e.g. [Connolly and Begg, 2005]). The *set-transaction* primitive enables using advanced non-default parameters for transaction management.

C A Simple JDBC Example

We end this subsection by an illustrative example of how an application can read and write² within the bookshop relational database, described previously and composed of the shelf and copy tables. The application is supposed to be written in Java (version 1.6). The application makes use of transaction primitives and benefits from the most general ability to use dynamic SQL, taking advantage of JDBC (version 4.0) [Oracle, 2014], the industry standard API for interaction between Java code and relational DBMSs.

The application must first import the JDBC API using a standard Java ‘import’:

```
import java.sql.*
```

¹This default behavior has been criticized as unsafe and some DBMSs rollback the transaction at normal termination. It is recommended to always perform an explicit commit at normal program termination.

²In general, an application can submit any SQL code to the DBMS, including notably DDL statements. Nevertheless, database engineering methodology often considers the database schema as static and manually defined.

then it must load the driver that offers the proprietary API code needed to interact with the specific DBMS implementation managing the database (here, the Oracle Express DBMS):

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

Once the API completely loaded, one can use it to get a connection to the database, providing the database network address and the necessary credentials:

```
String url = "jdbc:oracle:thin:@ " + "localhost:" + "1521:XE";
String login = "jean-luc";
String pwd = "sql";
Connection con = DriverManager.getConnection(url, login, pwd);
```

Using this connection, the following code reads an author name from the keyboard, extracts all the labels of shelves storing books from this author using a **SELECT** statement (a character string crafted using the read author name value), and prints the result rows using a **while** statement:

```
String author = getKeyboardInput();
String query = "SELECT DISTINCT label
               FROM shelf, copy
               WHERE shelf.identifier = copy.identifier
               AND copy.author = '"+author+"'";
ResultSet result = con.createStatement().executeQuery(query);

while (result.next()) {
    System.out.println(result.getInt("label"));
};

result.close();
```

One can notice in this code that the JDBC API provides the `executeQuery(String)` method to process **SELECT** statements over the database. The SQL code to execute must be provided as a character string parameter to be compiled at runtime³. The returned rows are stored in a **ResultSet** variable. The returned **ResultSet** object contains the list of rows in a random order (except if the **SELECT** statement had contained an **ORDER BY** clause) and a cursor pointing to the row considered the current one. When the **ResultSet** object is initialized, the cursor is pointing before the first row stored in the object. The `next()` method moves the cursor to the next row. It returns true when this move is successful and false when there are no more rows to read in the **ResultSet** object. The `next()` method is typically used as the loop condition of a **while** statement to iterate over the **ResultSet** object content. A column value for the current row can be retrieved using a `getType(String)` method over the **ResultSet** object, where **Type** represents the type of the concerned column and where the column name is provided as a string parameter. A call to the `close()` method over the **ResultSet** object releases the object and all its associated resources.

By default, JDBC automatically starts a new transaction before every single SQL statement and terminates this transaction once the statement fully executed. In the particular case of **SELECT** statements, the statement is considered as fully executed when the `close()`

³One should note that if a single SQL statement must be executed several times, it is more efficient to compile it only once. This can be done by using the `PreparedStatement` interface provided by the API.

method is called over the associate **ResultSet** object. This behavior is called auto-commit and can be disabled using the following method provided by the database connection:

```
con.setAutoCommit(false);
```

The code following this paragraph creates a new shelf as well as a new book copy in this shelf within the database. It reads the shelf and book details from the keyboard and uses the database connection to call the **execute(String)** method, which enables one to execute the **INSERT**, **DELETE** or **UPDATE** statement provided as a string parameter. The two **INSERT** statements are executed as a single transaction: the first **INSERT** initiates the transaction, the commit primitive is called after the successful termination of the two statements. If a failure occurs during the execution of the transaction, (for example, an integrity constraint of the database schema is violated by one of the two **INSERT** statements) the API warns the application by throwing an **SQLException**, which is caught to cancel the whole transaction (i.e. the two **INSERT** statements) using the rollback primitive:

```
int identifier = getKeyboardInput();
int label = getKeyboardInput();
int barcode = getKeyboardInput();
String title = getKeyboardInput();
String author = getKeyboardInput();
String insertShelf = "INSERT INTO shelf
                    VALUES (" + identifier + "," + label + ")";
String insertCopy = "INSERT INTO copy
                    VALUES (" + barcode + "," + title + "," + author + "," + identifier + ")";

try {
    con.createStatement().execute(insertShelf);
    con.createStatement().execute(insertCopy);
    con.commit();
} catch (SQLException e) {
    con.rollback();
};
```

When the application has finished working with the database, a call to the **close()** method over the **Connection** object releases the connection and all its associated resources:

```
con.close();
```

1.3 Alternatives to Relational Databases

If relational DBMSs have been very strong leaders on the database market for a number of decades, several alternate database technologies have been developed during the history of computer science. In this final section, we offer an historical perspective over the contenders of the relational model and of SQL. Starting with the old network and hierarchical data model of the 1960s, finally eclipsed by the relational model, we detail then the deductive, object and XML database initiatives, which were absorbed by relational DBMSs, and finish by presenting the recent criticism over the old dominance of relational DBMSs and the current NoSQL movement.

1960s-1970s: CODASYL and IMS

Before the introduction of the relational model in the early 1970s and the rise of SQL-based relational DBMSs in the 1980s, the emerging market of DBMSs was dominated by two major systems: CODASYL and IMS [Hainaut, 2012].

Founded in 1959, CODASYL (an acronym for Conference On DATA SYstem Languages) was a consortium of organizations grouping users and producers of data-oriented software, developing and promoting effective and standardized languages for computer-aided management of data systems. This consortium was notably responsible for the development of the COBOL programming language, introduced in 1960. In the late 1960s, the CODASYL group published a set of specifications for a DDL and DML based on the so-called *network model* of data. These specifications were implemented in many DBMSs and some of these CODASYL DBMSs are still in use nowadays, even if an SQL interface has typically been added to access the stored data.

In the network model, data is organized in a set of information units, called records and belongs to a given record type. A record is a group of scalar values aggregated according to a given structure. A record type both provides a domain grouping semantically correlated records and a common structure to be used by all the records of this type. Record types are associated through set types, comprising one owner record type and one or more member record type(s). For a defined set type associating owner type O with member types M_1 and M_2 , each record of type O in the database can be linked to one or more records of type M_1 and M_2 . Set types organize thus data in a network where nodes are records and edges are the links between these records. Records are retrieved and subsequently modified by locating particular records in a given type and by navigating along the edges of the network linking records together. A more advanced and detailed description of the network model can be found in [Silberschatz et al., 2010].

IMS is a proprietary DBMS that has been developed by IBM since 1965. It was originally built for the specific needs of the aerospace industry in the context of the Apollo program and then made available to a broader audience in 1969. IMS makes use of its own DDL/DML called DL/I (acronym for Data Language/Interface) based on the so-called *hierarchical model* of data. IMS is still supported by IBM nowadays and is still in use within high-availability software, notably in banking and insurances, but is often considered as legacy software. SQL interfaces have been developed to interact with IMS databases and IBM provides its own JDBC driver for IMS.

Conceptually, the hierarchical model organizes and enables access to data in a similar way

to the network model: pieces of information are stored in typed records (called segments) and the semantical relationships between data are represented by typed links connecting these segments, which should be followed to explore the data. The main difference is that the network model allows the network made of records and links to be an arbitrary graph, where the hierarchical model requires this network to be a set of trees, defining a hierarchy between their node segments. The trees are explored from root to leaves and complex rules enable jumping from one tree to another. IMS is a notoriously complex and irregular system. A more advanced and detailed description of the hierarchical model and of IMS can be found in [Silberschatz et al., 2010].

CODASYL DBMSs enable an efficient access to the stored data while requiring few hardware resources and offer a simple API. IMS is known as a very fast, powerful and reliable DBMS. The network and hierarchical model on which they are based are called *navigational* models, as reading and writing data into the database requires to write software applications that navigate through some linked lists of records, in order to retrieve data as well as to modify the content of the records and the structure of the lists.

1980s: Datalog

Datalog is a DDL and DML born from the will to merge relational database technology with artificial intelligence, in order to build so-called *deductive databases*. With deductive databases, one meant to provide DBMSs with the ability to make deductions over the stored data, i.e. to infer new facts from the facts present in the managed database. Datalog is both a simplified version of the Prolog logic programming language and an enriched version of the relational calculus introduced by Codd. Compared to the relational calculus, Datalog essentially introduces the ability to query the data in a recursive way. For example, if one considers a relational database storing genealogical information, basically describing parents-children relations between people, the relational algebra and calculus are not expressive enough to establish the list of ancestors of a person, as this will require to explore recursively the parents-children relations defined in the database. By providing relational-complete languages with recursion, Datalog, and its underlying execution mechanism based on logic programming, makes the DBMS able to answer such a query, deducing new facts from a recursive exploration of the existing facts. The interested reader can notably find a more detailed discussion of Datalog in [Abiteboul et al., 1995]. Initiated in 1977, the work over the integration of logic programming with relational databases exploded in the 1980s, as logic programming and SQL were very hot topics at that time. Despite this interest, Datalog was finally almost abandoned by researchers in the 1990s, notably considering the lack of interest in the industry, as famously stated in [Stonebraker and Hellerstein, 1998]: “No practical applications of recursive query theory [...] have been found to date”. Nevertheless, the ability to process recursive SQL queries was added to the SQL ISO standard in 1999, and Datalog has recently gained a renewed interest over new applications, like program analysis, network monitoring, etc [Huang et al., 2011].

1990s: Object Databases

Object-Oriented (OO) DBMSs have emerged as a consequence of the popularity of object-oriented programming and because of the so-called *impedance mismatch* between relational

DBMSs and object-oriented programming languages. On the one hand, OO languages are based on the principle of aggregating information in units of data, called objects and belonging to system or user-defined classes. A class simultaneously provides a domain grouping semantically correlated objects, a common structure for the data stored in the objects of the class (including other objects from given other classes) and a set of standard programmatic methods enabling other objects (belonging to authorized classes) to manipulate the data stored in objects of the class. Additional relations between classes can be defined using the mechanism of inheritance, enabling a class to inherit from the data structures and methods of another class. On the other hand, relational DBMSs enable one to manipulate information aggregated in rows and tables, using SQL, essentially according to the relational model. The impedance mismatch emphasizes the conceptual and technical overhead caused by the necessary translation between the object-oriented and relational models of data, when an object-oriented application needs to read or write data from a relational database. The main purpose of object databases is to remove this overhead by delivering DBMSs able to store data according to the object-oriented model and able to provide transparent persistence services for the objects manipulated using an OO language as well as for the relations existing between these objects. A more advanced and detailed description of object-oriented DBMSs can be found in [Connolly and Begg, 2005].

Originally developed in the 1980s, the golden age of OO DBMSs [Hainaut, 2012] is certainly the 1990s, with the relative success of several commercial products. The 1990s also saw the appearance of object-relational DBMSs, based on a data model integrating the relational model with object-oriented concepts. These efforts were incorporated (with varying levels of success) by the main relational DBMS manufacturers and adopted in the ISO SQL standard in 1999, as object-oriented extensions for SQL. These tactical moves stopped the rise of concurrent OO-based DBMSs and research in the topic subsequently fell in a phase of dormancy. The actual impact of OO database research is described as sparse [Hainaut, 2012].

Made popular by the appearance of the Hibernate project⁴, Object-Relational Mappings (ORMs) (see e.g. [O’Neil, 2008]) propose an alternate approach to deal with the object-relational impedance mismatch. In a nutshell, an ORM framework is a software system that enables object-oriented programmers to make their objects persistent in a seamless way, by handling transparently the storage of objects in an underlying relational database. ORMs have been strongly criticized in a famous blog entry⁵, where they are said to be the “Vietnam of computer science”, because they would provide a quick initial success, but would trap users into a situation requiring subsequent programming efforts which are not worth it.

2000s: XML databases

In the early 2000s, several commercial XML DBMSs have appeared (see e.g. [Hainaut, 2012] for a discussion), proposing to store information using the data model of XML documents (described as close to old hierarchical model) and to query the stored data using an ad-hoc DML. Similarly to what happened for object databases, relational DBMSs manufacturers have then offered the ability to store and manipulate XML-formatted data

⁴<http://hibernate.org>

⁵<http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx>

within relational databases, pulling the rug from under concurrent XML products.

2010s: NoSQL

More recently, several authors have defended [Stonebraker et al., 2007] a strong criticism over existing relational DBMSs products, accused of being too general and outdated: “the current R[elational] DBMS[s] [...] while attempting to be a “one size fits all” solution [...] excel at nothing [...] [and] should be retired in favor of a collection of “from scratch” specialized engines [...] design[ed] [...] for tomorrow’s requirements”.

At the same time, the database community has seen the emergence (see e.g. [Leavitt, 2010]) of a *NoSQL* movement. This movement was born with the recent rise of a new kind of businesses, offering services like internet social networking, online shopping and cloud computing. The specific technical needs of such businesses, requiring DBMSs able to scale, at minimal cost, to important amounts of concurrent accesses to huge amounts of data, massively distributed over a large number of servers, have justified the appearance of a wide range of ad-hoc non-relational DBMSs. Under the banner of NoSQL, they promote notably the use of a simpler DML (like a basic key/value storage) and the relaxing of the integrity controls offered by all relational DBMSs, in order to offer a better scalability and flexibility in the considered use cases, avoiding by the way the fee imposed by the licensing policies of the leading relational DBMS manufacturers. While NoSQL DBMSs have gained market shares, some authors have questioned the real efficiency of NoSQL DBMSs compared to SQL ones [Floratos et al., 2012], while other have argued that the relational model and ACID properties cannot be responsible for a supposed lack of performance observed in relational DBMSs [Stonebraker, 2010]. These last authors have pled for the design of scalability-oriented relational DBMSs, still offering SQL and ACID properties, under the banner of NewSQL [Stonebraker, 2012].

Chapter 2

SAT, SMT and Relational Solvers

Contents

2.1	SAT Solvers	47
2.1.1	The SAT Problem	47
2.1.2	Complexity of SAT and Practical Applications	48
2.1.3	The DPLL Algorithm for SAT	48
2.1.4	Alternative Algorithms for SAT	53
2.2	SMT Solvers	55
2.2.1	The SMT Problem	55
2.2.2	The SMT-Lib Language	57
2.2.3	Complexity of SMT and Practical Applications	59
2.2.4	The DPLL(T) Algorithm for SMT	60
2.2.5	The Nelson-Oppen Method for Theory Combination	64
2.2.6	Handling Quantifiers	65
2.2.7	Challenges and Alternatives	72
2.3	Relational Solvers	74
2.3.1	Writing and Solving Relational Constraints	74
2.3.2	The Alloy Framework	74
2.3.3	From Alloy to SMT	79

CONSTRAINT programming is a programming paradigm where the problem to solve is expressed as a set of constraints to satisfy. Constraint programming frameworks are typically based over two main components: a *constraint language* and a *constraint solver*. The language defines a standard formal syntax to express the constraints, while the solver offers algorithms to solve them. Logic is a common language used in constraint programming frameworks. The constraints are written as a logical theory and solving the constraints means finding a satisfying model for the theory.

In this thesis, we generalize an existing software testing technique based on constraint programming, in order to test applications involving SQL code. Constraints and logic constitute the Rosetta stone of our generalization. Given that SQL is based over relational algebra and thus calculus, the behavior of SQL code can be described using first-order logic. This logical description can be seen as a set of logical constraints over the content of the manipulated database relational tables. These logical constraints over relational objects can be integrated within the logical constraints generated by classical constraint-based testing techniques, making these techniques able to handle SQL code.

In this chapter, we review the two main existing constraint frameworks based on logic: *SAT* (where the constraint language is propositional logic) and *SMT* (where the constraint language is first-order logic, enriched with some kind of datatype operator symbols). We conclude by showing how these frameworks have already been used in the literature to solve constraints involving relational objects, as those produced in this work.

In section 2.1, we describe the SAT framework, by presenting propositional logic and detailing the *DPLL algorithm*, which is at the core of most SAT solvers. The SMT framework is presented in section 2.2. First, we remind *first-order logic satisfiability* and show how it can be enforced *modulo some Σ -theories*. Then, we present the standard *SMT-Lib language* for writing SMT instances and show how the DPLL algorithm from SAT can be extended to handle the full SMT problem. This enables us to introduce the main components of state-of-art modern SMT solvers: the *DPLL(T) algorithm*, the *Nelson-Oppen method* for theory combination and the main *quantifier handling* techniques. In section 2.3, the *Alloy framework* for relational constraint solving is discussed: the Alloy constraint language is presented and we show how SAT and SMT solvers have been used to solve Alloy constraints.

This chapter proposes an essentially operational view of the presented constraint programming techniques. It presents the main principles of each constraint framework and introduces the necessary context to understand the practical use of these frameworks, such as in the research work presented in this thesis.

2.1 SAT Solvers

2.1.1 The SAT Problem

Propositional logic [Büning and Lettmann, 1999, Malik and Zhang, 2009] is a formal language that enables one to define logical formulas. It is a subset of the first-order logic introduced in the previous chapter (page 18). A sentence written in first-order logic is also a sentence of propositional logic if and only if it contains no quantifier and no predicate, function, variable or non-boolean constant symbols. The syntax of propositional logic is thus given by the following BNF:

$$\begin{aligned} \langle theory \rangle &::= \{ \text{FORMULA } \langle formula \rangle ; \}^* \\ \langle formula \rangle &::= \langle symbol \rangle && \text{(Boolean Constant)} \\ &| \neg \langle formula \rangle && \text{(Logical Not)} \\ &| \langle formula \rangle \wedge \langle formula \rangle && \text{(Logical And)} \\ &| \langle formula \rangle \vee \langle formula \rangle && \text{(Logical Or)} \\ &| \langle formula \rangle \implies \langle formula \rangle && \text{(Implication)} \\ &| \langle formula \rangle \iff \langle formula \rangle && \text{(Equivalence)} \\ &| (\langle formula \rangle) \end{aligned}$$

An example of a propositional theory, involving the boolean constants b_1 and b_2 , is:

FORMULA $b_2 \wedge (\neg b_2 \implies b_1) \wedge b_1$

Propositional satisfiability is first-order logic satisfiability restricted to propositional theories. A model for a theory in propositional logic associates each constant symbol with a corresponding boolean value true or false. A theory is satisfied by a model if all the formulas in the logic are true considering the values provided by the model. This enables us to define the SAT problem:

Definition 2.1. *Given a theory in propositional logic, the propositional **SAT**isfiability **problem**, abbreviated as **SAT**, is the problem of finding a model satisfying the theory or proving that none exists.*

For example, the model $\{b_1 = \text{true}, b_2 = \text{true}\}$ satisfies the example theory given above, as its only formula becomes

$$\text{true} \wedge (\neg \text{true} \implies \text{true}) \wedge \text{true}$$

which is true. On the contrary, the following theory obviously admits no satisfying model and is thus said to be *unsatisfiable*:

FORMULA $b_1 \wedge (\neg b_1)$

The SAT problem is computable [Malik and Zhang, 2009] and we can define SAT solvers:

Definition 2.2. *A **SAT solver** is a computation procedure for the SAT problem. For any instance of the SAT problem, a SAT solver either returns a satisfying model or establishes that the considered theory is unsatisfiable.*

Many SAT solvers have been developed and are regularly benchmarked during a community-driven competition held with the International Conference on Theory and Applications of Satisfiability [Malik and Zhang, 2009]. Examples of popular [SAT-Association, 2014] open-source SAT solvers are Minisat [Eén and Sörensson, 2003] (C++), Picosat [Biere, 2008] (C) and SAT4J [Berre and Parrain, 2010] (Java).

2.1.2 Complexity of SAT and Practical Applications

The Cook theorem [Cook, 1971] establishes that the SAT problem belongs to the class of NP-complete problems [Garey and Johnson, 1979]. Any instance of a problem in this class can be transformed into an instance of any other problem of the class, in polynomial time. Up to now, all the known procedures solving an NP-complete problem necessitate, for some of the problem instances, an execution time that grows exponentially with the size of the considered instance. It is widely supposed (but not demonstrated) that there is no procedure able to solve all instances of an NP-complete problem in sub-exponential time. This strong conjecture makes the hardest instances of NP-Complete problems intractable using exhaustive methods.

Many real-world problems can be encoded and solved as SAT instances [Malik and Zhang, 2009]. The key idea is to express the problem as a set of *constraints* over boolean values to be satisfied. If the constraints are written as a propositional theory, a SAT solver can be used to search for a solution to these constraints and in this way solve the problem.

However, the NP-completeness of SAT is a serious theoretical obstacle to the applicability of SAT solvers over real-world problems. In practice, decades of progress in SAT solver technologies and in computer speed enable current solvers to handle SAT instances involving more than one million of boolean constants [Malik and Zhang, 2009]. But, as it can be expected from the characterization of the problem as NP-complete, they do not scale over worst case instances. Nevertheless, many real-world SAT instances do not belong to worst case and SAT solvers have been successfully applied in many practical applications, like design of digital circuits [Nam et al., 1999], planning [Kautz et al., 1992], model-checking [Biere et al., 1999], etc.

2.1.3 The DPLL Algorithm for SAT

A Algorithms for SAT

Several design approaches have been considered and studied for building SAT solvers. The first proposed algorithm is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Davis et al., 1962, Davis and Putnam, 1960]. This algorithm remains very popular despite its old age and is of strong theoretical importance [Malik and Zhang, 2009]. Moreover, this algorithm is the underlying basis of most SMT solvers, introduced in the next section. We detail the DPLL algorithm in this subsection and present briefly the main other SAT solving algorithms in the next subsection.

B Conjunctive Normal Form

Conjunctive Normal Form (CNF) is a restricted syntax for propositional theories. A propositional formula is a *literal* if and only if it is either a boolean constant or the negation of a boolean constant. A propositional formula is a *clause* if and only if it is a disjunction (i.e. logical OR) of literals. A propositional formula is a *formula in CNF* if and only if it is a conjunction (i.e. logical AND) of clauses, i.e. if and only if it is a sentence of the following grammar:

$$\begin{aligned} \langle \text{cnf-formula} \rangle &::= (\langle \text{clause} \rangle) && \text{(Clause)} \\ | \quad \langle \text{cnf-formula} \rangle \wedge \langle \text{cnf-formula} \rangle && \text{(Logical And)} \end{aligned}$$

$\langle clause \rangle ::= \langle literal \rangle$	(Literal)
$\quad \quad \langle clause \rangle \vee \langle clause \rangle$	(Logical Or)
$\langle literal \rangle ::= \langle symbol \rangle$	(Boolean Constant)
$\quad \quad \neg \langle symbol \rangle$	(Negation of Boolean Constant)

For example, the following formula is in CNF, as it is an AND of ORs of literals:

$$(\neg b_1) \wedge (\neg b_3 \vee b_1 \vee \neg b_3) \wedge (\neg b_1 \vee b_3)$$

The expressive power of CNF formulas is equivalent to the expressive power of the whole propositional logic:

Theorem 2.1. *Every theory in propositional logic can be transformed into an equivalent CNF formula.*

with

Definition 2.3. *Two logical theories are **equivalent** if they have the same truth value in every possible model.*

A detailed CNF transformation algorithm can be found in [Büning and Lettmann, 1999]. It can be shown that, in the worst case, the CNF transformation increases exponentially the size of the input formula [Büning and Lettmann, 1999]. The Tseitin's CNF transformation [Tseitin, 1983] makes it possible to guarantee a linear increase of the formula's size, but it introduces new boolean constants in the formula, producing an equisatisfiable [Büning and Lettmann, 1999] but possibly not equivalent CNF formula:

Definition 2.4. *Two logical theories sharing symbols are **equisatisfiable** if, for every model that satisfies one of the theories, there exists at least one model providing the same interpretation to the shared symbols which satisfies the other theory.*

For example, the two following theories are not equivalent but equisatisfiable:

FORMULA b_1

FORMULA $b_1 \wedge b_2$

Indeed, as the two theories have not the same truth value considering the model $\{b_1 = true, b_2 = false\}$, they are not equivalent. However, as $\{b_1 = true, b_2 = false\}$ and $\{b_1 = true, b_2 = true\}$ are the only models that satisfy the first theory and $\{b_1 = true, b_2 = true\}$ the only model that satisfies the second theory, all the satisfying models for both theories assign true to the b_1 shared symbol, so that the two theories are equisatisfiable.

C The DPLL Algorithm

The DPLL algorithm can only find a satisfying model or establish unsatisfiability of a CNF formula. The input theory must thus be preprocessed to be transformed into an equisatisfiable CNF formula.

As an illustrating example, we show how the DPLL algorithm solves the SAT problem over a sample SAT instance, defined by the following theory:

FORMULA $(b_1 \vee b_3)$

FORMULA $(b_2 \wedge \neg b_1) \vee (b_6 \wedge \neg b_1) \vee (b_2 \wedge b_4) \vee (b_6 \wedge b_4)$

FORMULA $\neg(b_3 \wedge b_4 \wedge b_5)$

FORMULA $b_3 \iff b_5$

FORMULA $(b_3 \vee b_5)$

In this sample theory, the first and last formulas are already in CNF. The second formula can be transformed into CNF using the boolean distributivity rule:

$$\begin{aligned} (b_2 \wedge \neg b_1) \vee (b_6 \wedge \neg b_1) \vee (b_2 \wedge b_4) \vee (b_6 \wedge b_4) &\leftrightarrow ((b_2 \vee b_6) \wedge \neg b_1) \vee ((b_2 \vee b_6) \wedge b_4) \\ &\leftrightarrow (b_2 \vee b_6) \wedge (\neg b_1 \vee b_4) \end{aligned}$$

The third formula can be transformed using the De Morgan's laws:

$$\neg(b_3 \wedge b_4 \wedge b_5) \leftrightarrow (\neg b_3 \vee \neg b_4 \vee \neg b_5)$$

And the fourth formula can be transformed by expressing $\iff$ in terms of $\wedge$, $\vee$ and $\neg$:

$$\begin{aligned} b_3 \iff b_5 &\leftrightarrow (b_3 \implies b_5) \wedge (b_5 \implies b_3) \\ &\leftrightarrow (\neg b_3 \vee b_5) \wedge (\neg b_5 \vee b_3) \end{aligned}$$

The resulting CNF version of the theory is thus:

FORMULA $(b_1 \vee b_3)$

FORMULA $(b_2 \vee b_6) \wedge (\neg b_1 \vee b_4)$

FORMULA $(\neg b_3 \vee \neg b_4 \vee \neg b_5)$

FORMULA $(\neg b_3 \vee b_5) \wedge (b_3 \vee \neg b_5)$

FORMULA $(b_3 \vee b_5)$

Once each formula in CNF, the theory can be transformed into an equivalent (or equisatisfiable) single CNF formula: the conjunction of its formulas. Indeed, given the definition of theory satisfiability, a theory is always equivalent to the single formula made of the conjunction of its own formulas. If all the theory's formulas are in CNF, their conjunction is in CNF as well.

In particular, the sample theory considered in our illustrative example is thus equivalent to the following CNF formula:

$$\underbrace{(b_1 \vee b_3)}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{(\neg b_1 \vee b_4)}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee \neg b_4 \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}}$$

The DPLL algorithm can then be applied over the obtained CNF formula. From a general standpoint, the algorithm starts by selecting randomly a boolean constant among those defined in the formula and assigns it a value. Then, the algorithm deduces the consequences of this assignment, considering recursively the following rules:

1. If one literal in a clause is true then the clause is true.
2. If all the literals in a clause are false except one whose value is unknown, this last literal must be true for the formula to be true

Once the consequences computed, the process is repeated over the obtained formula.

In the formula considered here, the algorithm can select for example constant b_1 and assign *true* to it. b_1 appears in clauses 1 and 3 and the consequences of b_1 being true are that clause 1 is true and that b_4 should be true for clause 3 to be true. The algorithm then computes recursively the consequence of b_4 being true. b_4 appears only in clause 4 from which nothing can be concluded. The resulting formula is:

$$\underbrace{true}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{true}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee false \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}}$$

The process is then repeated over this formula, by example by assigning true to b_2 , which makes clause 2 true and produces the following formula:

$$\underbrace{true}_{\text{clause 1}} \wedge \underbrace{true}_{\text{clause 2}} \wedge \underbrace{true}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee false \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}}$$

Similarly, true can be assigned to b_3 , making clause 6 and 7 true, but causing a *conflict* as clause 4 requires b_5 to be false and clause 5 requires b_5 to be true. This conflict means that no model where $b_1 = true$, $b_4 = true$, $b_2 = true$ and $b_3 = true$ can satisfy the formula.

When the DPLL algorithm detects a conflict, it backtracks to the last assigned constant for which both choices have not been tried, undoes all the subsequent constant assignments, flips the constant's value and restarts the process.

In the example considered here, the last assigned constant for which both choices have not been tried is b_3 . The process restarts over the following formula and assigns b_3 to false:

$$\underbrace{true}_{\text{clause 1}} \wedge \underbrace{true}_{\text{clause 2}} \wedge \underbrace{true}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee false \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}}$$

The assignment of false to b_3 leads to a conflict as well, as clause 6 requires b_5 to be false and clause 7 requires b_5 to be true. The last assigned constant for which both choices have not been tried is b_2 . The process restarts over the following formula and assigns false to b_2 :

$$\underbrace{true}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{true}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee false \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}}$$

b_6 must be false and the resulting formula is:

$$\underbrace{true}_{\text{clause 1}} \wedge \underbrace{true}_{\text{clause 2}} \wedge \underbrace{true}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee false \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}}$$

Assigning b_3 both to true and false will lead to a conflict in a similar way than before, and the algorithm will backtrack over the initial formula to assign false to b_1 :

$$\underbrace{(b_1 \vee b_3)}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{(\neg b_1 \vee b_4)}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee \neg b_4 \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}}$$

As b_1 is false, clause 3 will be true, while clause 1 will require b_3 to be true. Considering b_3 to be true, clause 5 will require b_5 to be true and clauses 6 and 7 will be true. Considering b_5 (and b_3) to be true, clause 4 will require b_4 to be false, from which nothing can be concluded. The resulting formula will be:

$$\underbrace{true}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{true}_{\text{clause 3}} \wedge \underbrace{true}_{\text{clause 4}} \wedge \underbrace{true}_{\text{clause 5}} \wedge \underbrace{true}_{\text{clause 6}} \wedge \underbrace{true}_{\text{clause 7}}$$

It remains for the algorithm to assign true to b_2 or b_6 to find a satisfying model for the formula, like for example:

$$\{b_1 = \text{false}, b_2 = \text{true}, b_3 = \text{true}, b_4 = \text{false}, b_5 = \text{true}, b_6 = \text{false}\}$$

The SAT problem can be considered as a search problem whose search space is the set of all possible constant assignments. The constant assignment and backtracking parts of the DPLL algorithm basically perform a systematic search of this search space. As the number of possible assignments grows exponentially with the number of constants, the consequence deduction part of the algorithm tries to prune away the search space, avoiding as much as possible a complete exploration of all the possible models. In accordance with the problem being NP-complete, an exponential number of models must still be explored for worst case instances. If no satisfying assignment is found, the algorithm returns that the instance is unsatisfiable.

D Clause Learning, Backjumping and other Improvements

The DPLL algorithm has been proposed in the beginning of the 1960s and many techniques improving its efficiency have been developed since then. These improvements notably include preprocessing of the input formula to eliminate clauses and constants [Eén and Biere, 2005], efficient implementation for the consequence deduction mechanism [Moskewicz et al., 2001], heuristic choice of the assigned literal [Moskewicz et al., 2001, Silva, 1999], parallel implementation [Hamadi et al., 2009] and random restart of the search process for a wider exploration of the search-space [Gomes et al., 1998]. More importantly, conflict-driven *clause learning* (CDCL) and *backjumping* [Marques Silva and Sakallah, 1996] have been among the most effective proposed DPLL improvements and constitute a key part of most current SAT solvers [Malik and Zhang, 2009].

Clause learning and backjumping basically improve the behavior of the DPLL algorithm when a conflict is detected. With clause learning, the conflict is analyzed in order to add a new clause, called a learned clause, to the formula, which prevents the algorithm from making assignments leading again to the same conflict. The learned clause is built so that it is always redundant with the original formula, improving the solving process without invalidating the solution. Backjumping uses the same conflict analysis information during the backtracking step, in order to jump back directly before the earliest assignment that implies the conflict. Backjumping and clause learning enable one to prune away an unsatisfiable region of the search space.

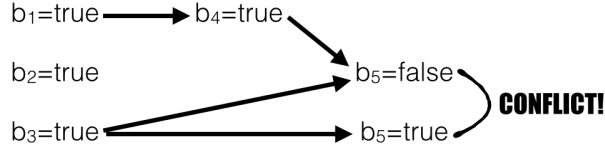
We now demonstrate the DPLL algorithm with clause learning and backjumping over the sample CNF formula considered before:

$$\underbrace{(b_1 \vee b_3)}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{(\neg b_1 \vee b_4)}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee \neg b_4 \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}}$$

As previously, the algorithm assigns true to b_1 , requiring b_4 to be true as well, and then it assigns true to b_2 and b_3 , leading again to the conflict between clause 4 (where the values of b_3 and b_4 require b_5 to be false) and 5 (where the value of b_3 requires b_5 to be true):

$$\underbrace{\text{true}}_{\text{clause 1}} \wedge \underbrace{\text{true}}_{\text{clause 2}} \wedge \underbrace{\text{true}}_{\text{clause 3}} \wedge \underbrace{(\text{false} \vee \text{false} \vee \neg b_5)}_{\text{clause 4}} \wedge \underbrace{(\text{false} \vee b_5)}_{\text{clause 5}} \wedge \underbrace{\text{true}}_{\text{clause 6}} \wedge \underbrace{\text{true}}_{\text{clause 7}}$$

The conflict analysis is performed by building and processing a graph of the constant assignments made so far and of the implications between them:



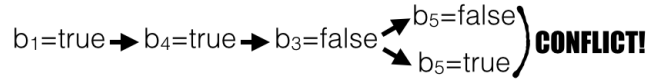
From this graph, one can deduce that the conflict will occur whenever b_1 and b_3 are true at the same time. The algorithm then backjumps before the assignment of b_1 to true and the learned clause $(\neg b_1 \vee \neg b_3)$ is added to the formula. The DPLL algorithm then restarts normally over the following formula:

$$\underbrace{(b_1 \vee b_3)}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{(\neg b_1 \vee b_4)}_{\text{clause 3}} \wedge \underbrace{(\neg b_3 \vee \neg b_4 \vee \neg b_5)}_{\text{clause 4}} \\
 \wedge \underbrace{(\neg b_3 \vee b_5)}_{\text{clause 5}} \wedge \underbrace{(b_3 \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(b_3 \vee b_5)}_{\text{clause 7}} \wedge \underbrace{(\neg b_1 \vee \neg b_3)}_{\text{learned clause 1}}$$

True is assigned to b_1 , which still implies that b_4 must be true, which implies that b_3 should be false because of the learned clause. This leads to a conflict between clause 6 (where the value of b_3 requires b_5 to be true) and 7 (where the value of b_3 requires b_5 to be false):

$$\underbrace{\text{true}}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{\text{true}}_{\text{clause 3}} \wedge \underbrace{\text{true}}_{\text{clause 4}} \wedge \underbrace{\text{true}}_{\text{clause 5}} \wedge \underbrace{(\text{false} \vee \neg b_5)}_{\text{clause 6}} \wedge \underbrace{(\text{false} \vee b_5)}_{\text{clause 7}} \wedge \underbrace{\text{true}}_{\text{learned clause 1}}$$

The conflict analysis graph is:



which means that a conflict will occur whenever b_1 will be true. The algorithm then backtracks and assigns definitely false to b_1 . As previously, this will require b_3 and b_5 to be true and b_4 to be false, leading to the following formula:

$$\underbrace{\text{true}}_{\text{clause 1}} \wedge \underbrace{(b_2 \vee b_6)}_{\text{clause 2}} \wedge \underbrace{\text{true}}_{\text{clause 3}} \wedge \underbrace{\text{true}}_{\text{clause 4}} \wedge \underbrace{\text{true}}_{\text{clause 5}} \wedge \underbrace{\text{true}}_{\text{clause 6}} \wedge \underbrace{\text{true}}_{\text{clause 7}} \wedge \underbrace{\text{true}}_{\text{learned clause 1}}$$

It remains again for the algorithm to assign true to b_2 or b_6 to find a satisfying model for the formula. Whereas there are 64 possible assignments for the set of 6 constants in the considered formula, the original DPLL algorithm was able to find a satisfying one in 5 trials. Thanks to a more sophisticated pruning mechanism, the DPLL algorithm augmented with clause learning and backjumping was able to find this satisfying assignment in only 3 trials.

2.1.4 Alternative Algorithms for SAT

If most SAT solvers are based on the DPLL algorithm [Malik and Zhang, 2009], many other exhaustive approaches have been developed to solve the SAT problem. Among these, some (e.g. [Puri and Gu, 1996]) make use of Binary Decision Diagrams (BDD) [Bryant, 1986], which offer a compact representation of a SAT formula in terms of its satisfying assignments. The Stålmarck's algorithm [Stålmarck, 1994] constitute a proprietary variant of the DPLL approach and has been applied successfully in some applications.

If exhaustive solvers have been successful in many practical applications, many kinds of SAT instances still remain hard to solve. In order for dealing with such instances where exhaustive approaches fail scaling, solvers based on heuristics have been developed. The considered heuristics are either common search heuristics, like genetic algorithms,

neural networks and simulated annealing [Marchiori and Rossi, 1999, Spears, 1993, Spears, 1996], ad-hoc local search [Selman et al., 1992] or survey propagation [Braunstein et al., 2005], a search heuristic introduced in by the statistical physics community. As they do not explore exhaustively the search-space, these techniques cannot guarantee finding a satisfying assignment for any satisfiable input instance. For the same reasons, they cannot establish that a given instance is unsatisfiable.

2.2 SMT Solvers

2.2.1 The SMT Problem

The SAT problem can be naturally generalized to first-order satisfiability:

Definition 2.5. *Given a theory written in first-order logic, the **first-order satisfiability problem** is the problem of finding a model satisfying the theory or proving that none exists.*

A sample instance of this problem is finding a satisfying model for:

```
CONSTANT 0 ON  $\mathbb{Z}$ ;  
FUNCTION  $-$  FROM  $\mathbb{Z} \times \mathbb{Z}$  TO  $\mathbb{Z}$ ;  
PREDICATE  $\leq$  ON  $\mathbb{Z} \times \mathbb{Z}$ ;  
VARIABLE  $a$  ON  $\mathbb{Z}$ ;  
CONSTANT  $b$  ON  $\mathbb{Z}$ ;  
FORMULA  $\exists a, ((a = 0) \wedge (a - b \leq 0))$ ;
```

Considering only models with the standard integer arithmetic interpretation of symbols $\mathbb{Z}$, 0 , $-$ and $\leq$, the theory is satisfiable: one can assign 0 to variable a and use a model where constant b is a non-strictly positive integer. Nevertheless, the theory is also satisfiable considering other models, with non-standard interpretations of the symbols $\mathbb{Z}$, 0 , $-$ and $\leq$. For example, considering the models where $\mathbb{Z}$ is still the set of integers, but where $-$ is a function that returns its first argument, $\leq$ the predicate that always returns true and 0 the integer 5 , the theory is satisfiable as one just needs to assign the integer 5 to variable a with a model where constant b is any object.

In practical applications, non-standard models are often totally irrelevant. As a consequence, one may need to restrict the satisfiability search to a set of models sharing a standard interpretation for some given symbols. Such an interpretation is usually called a Σ -theory:

Definition 2.6. *A Σ -theory is a group of constant, function and predicate symbols declared over some set symbols and a semantic value assignment (also called interpretation) for every of these constant, function, predicate and set symbols.*

A Σ -theory typically defines a unique set symbol, describing a particular datatype (like integers, reals, arrays, strings, etc.), a group of constant symbols to represent the datatype instances (like integer, real, array or string values) and a group of function and predicate symbols to represent the common operators defined over the datatype (like the classical “+” operator over integers or reals, the element read/write operators over arrays or the string length operator).

Several Σ -theories are compatible if they assign the same semantic value to every symbol that they share. We have then:

Definition 2.7. *The **combination of several compatible Σ -theories** is the Σ -theory composed of the union of all the symbols proposed by the original Σ -theories with their associated semantic values.*

A model enforces a Σ -theory if the model and the Σ -theory both assign the same semantic value to every symbol that they share. We have then [Barrett et al., 2009]:

Definition 2.8. *Given a theory in first-order logic together with a combination of zero, one or more compatible Σ -theories, the **Satisfiability Modulo Theories problem**, abbreviated as **SMT**, is the problem of finding a model satisfying the theory, among those enforcing the combination of Σ -theories, or proving that none exists.*

As an example, the following set of symbol declarations

CONSTANT 0 **ON** $\mathbb{Z}$;
CONSTANT 1 **ON** $\mathbb{Z}$;
FUNCTION + **FROM** $\mathbb{Z} \times \mathbb{Z}$ **TO** $\mathbb{Z}$;
FUNCTION − **FROM** $\mathbb{Z} \times \mathbb{Z}$ **TO** $\mathbb{Z}$;
PREDICATE $\leq$ **ON** $\mathbb{Z} \times \mathbb{Z}$;

and their classical integer arithmetic interpretation is called the Presburger arithmetic Σ -theory [Presburger, 1931, Barrett et al., 2009]. Searching for a satisfying model of the sample theory that we have just considered at the beginning of this section, among the models enforcing the Presburger arithmetic Σ -theory, is an instance of the SMT problem. This instance is satisfiable by any model where constant b is a non-strictly positive integer. The satisfiability search is then said to have been achieved modulo Presburger arithmetic.

We can now introduce the concept of an SMT *logic*:

Definition 2.9. *A **logic** is a couple made of a sublanguage of first-order logic and a combination of zero, one or more compatible Σ -theories.*

A logic defines a subset of SMT instances: an SMT instance is part of a logic if the instance's first-order theory is part of the logic's sublanguage and if the instance involves the same combination of Σ -theories as the logic. For example, if the logic is composed of the propositional part of first-order logic and of an empty combination of Σ -theories, the resulting subset of SMT instances is the set of SAT instances. SAT is thus also a subproblem of SMT, as is general first-order satisfiability. Similarly, if the logic is composed of conjunctions of predicates involving only uninterpreted integer constants and symbols from the linear integer arithmetic Σ -theory, the resulting subset of SMT instances is the set of linear integer programming instances [Papadimitriou, 1981].

Some logics are not computable, i.e. the problem of finding a satisfying model (or proving that none exists) for any SMT instance within the logic is not computable. Example of undecidable logics are first-order logic [Church, 1936, Turing, 1936] or non-linear integer programming [Gödel, 1931, Paris and Harrington, 1977]. The SMT problem is thus not computable in general, even if many logics are computable. For example, linear integer programming [Presburger, 1931, Cooper, 1972] and many fragments of first-order logic [Hustadt et al., 2004], including propositional logic, are computable. Partial procedures have often been developed for handling undecidable logics (see e.g. [Reynolds et al., 2013] for first-order logic and [Jovanovic and de Moura, 2012] for non-linear integer programming).

We can now define SMT solvers:

Definition 2.10. *An **SMT solver** is a partial computation procedure for the SMT problem. For some instances of the SMT problem, a SMT solver either returns a satisfying model within those enforcing the considered Σ -theories or establishes that the considered theory is unsatisfiable modulo these Σ -theories.*

Many SMT solving engines have been developed. Each of these solvers can handle more or less efficiently a greater or lesser number of decidable logics, as well as a greater or lesser number of instances among a greater or lesser number of undecidable logics. Coverage and efficiency of existing solvers are regularly benchmarked during a community-driven competition held with the International Conference on Theory and Applications of Satisfiability [De Moura and Bjørner, 2011]. A recent and extensive list of the benchmarked solvers and related details can be found in [Barrett et al., 2013].

2.2.2 The SMT-Lib Language

SMT-Lib [Barrett et al., 2010] is a language developed by the SMT research community for writing SMT instances. It is used as a standard input language by many SMT solvers, even if some dialectic differences remain, especially for non-standard Σ -theories. An SMT instance written in SMT-Lib is mainly composed of the declaration of a logic, followed by a theory declaration. The declaration of the logic establishes the logic to which the instance belongs and indicates thus to the solver which symbol interpretations to use for handling this SMT instance. The theory declaration details the first-order theory for which a satisfying model should be found, among those enforcing the combination of Σ -theories declared in the logic. We briefly present these two main components in the remainder of this subsection. A detailed description of the SMT-Lib language can be found in [Barrett et al., 2010]. As SMT-Lib is the community standard in SMT, we will use it to write SMT instances in the remaining part of this work.

The declaration of the logic is made using the **set—logic** keyword followed by the logic name. A list grouping examples of common logic names is presented in the following table. Each name is accompanied by (and composed of the initials of) the sublanguage of first-order syntax and the collection of Σ -theories defining the designated logic. The exact definition of the symbols offered by every available Σ -theory is given either in the SMT-Lib specification [Barrett et al., 2010] for standard Σ -theories (like integers, reals, arrays or bit-vectors), or by the SMT solver developers for non-standard Σ -theories. A logic is said to admit uninterpreted functions if it allows to write formulas containing set, predicate and function symbols that are not part of the Σ -theories considered within the logic, as well as variable and uninterpreted constant symbols that do not belong the sets defined by these Σ -theories. The solver must then find a value for all these uninterpreted symbols as a part of the returned model. If uninterpreted functions are not allowed, all the symbols appearing in the instance’s formulas must be associated to the Σ -theories considered within the logic.

Logic Name	Part of First-Order Logic	Σ -theories
QF_UF	Quantifier-Free propositional logic with Uninterpreted Functions	None
QF_LIA	Quantifier-Free propositional logic without uninterpreted functions	Linear Integer Arithmetic
QF_UFLIA	Quantifier-Free propositional logic with Uninterpreted Functions	Linear Integer Arithmetic
QF_AUFLIA	Quantifier-Free propositional logic with Uninterpreted Functions	Linear Integer Arithmetic and Arrays

AUFLIA	full first-order logic (with Uninterpreted Functions)	Linear Integer Arithmetic and Arrays
AUFNIRA	full first-order logic (with Uninterpreted Functions)	Non-linear Integer and Real Arithmetic and Arrays

The theory declaration is made using a particular syntax for first-order logic. As an example of this syntax, let us consider the following first-order theory, for which a satisfying model should be found:

PREDICATE P **ON** $S_1 \times S_1$;
VARIABLE x **ON** S_1 ;
VARIABLE y **ON** S_1 ;
FORMULA $\forall y, \exists x, P(x, y)$;
FUNCTION F **FROM** S_1 **TO** S_1 ;
CONSTANT b **ON** B;
FORMULA $((b \wedge P(x, x)) \vee \neg b) \iff (b \implies (F(x) = y))$;

This theory is expressed in SMT-Lib as:

```
(declare-sort S1)
(declare-fun P (S1 S1) Bool)
(assert (forall ((y S1)) (exists ((x S1)) (P x y))))
(declare-fun F (S1) S1)
(declare-const b (Bool))
(assert (forall ((x S1) (y S1)) (= (or (and b (P x x)) (not b)) (= b (= (F x) y)))))
```

This example has been chosen to illustrate the main differences between SMT-Lib and the notation previously used here for first-order logic. In SMT-Lib, symbols are declared using the **declare** keyword. Set symbols must be explicitly declared using the **declare-sort** keyword. Predicate symbols must be declared as boolean function symbols. Formulas are declared using the **assert** keyword. Free variables cannot be used directly and must be translated into universally quantified variables. Expressions used in formulas obey the nested syntax of S-expressions [McCarthy, 1960] popularized by the Lisp language (where, for example, “ $1 + 1 + 1$ ” is written “ $(+ (+ 1 1) 1)$ ”). Logical equivalence is expressed using the boolean equality (“ $=$ ”) symbol.

The symbols declared in the Σ -theories made available by the declaration of the logic should naturally be used within the theory. For example, in the following SMT instance, the declared logic is QF_LIA. The theory will thus not contain any quantifier. Moreover, as uninterpreted functions are not allowed, all the set, function and predicate symbols in the theory will be those declared in the Linear Integer Arithmetic Σ -theory, as defined in the SMT-Lib specification. These are the **Int** symbol, representing the set of integers, the integer constants, and the classical linear arithmetic operators and predicates (“ $+$ ”, “ $-$ ”, “ $\leq$ ”, “ $<$ ”, “ $\geq$ ”, “ $>$ ”) with their standard interpretation:

```
(set-logic QF_LIA)
(declare-const a (Int))
(declare-const b (Int))
(assert (and (= a 0) (<= (- a b) 0)))
```

If an SMT solver capable of handling the QF_LIA logic is given this SMT instance, it will search for a satisfying model modulo linear arithmetic. Solutions will be models where a is 0 and b a non-strictly positive integer.

The AUFLIA logic allows quantifiers and uninterpreted functions. It also allows to use both the symbols of the Linear Integer Arithmetic Σ -theory and of the Array Σ -theory, defined in the SMT-Lib specification. The Array Σ -theory provides the set symbol “(**Array Int Int**)” (for the set of arrays with integer indices and elements) and the function symbols **select** and **store**. The **select** $a\ i$ function returns the integer stored at index i in the array a . The **store** $a\ i\ e$ function returns the array obtained by storing the integer e at index i in the array a . An example of an SMT instance using the expressive power offered by the AUFLIA logic is:

```
(set-logic AUFLIA)
(declare-fun f (Int) Int)
(declare-const a (Array Int Int))
(assert (forall ((i Int)) (= a (store a (+ i 1) (f (select a i))))))
```

As solver providers develop solving capabilities for new Σ -theories, they often provide ad-hoc extensions for the SMT-Lib language. As an example, the character string Σ -theory recently provided by solver CVC4 [Barrett et al., 2011] offers non-standard symbols [CVC4, 2014] for writing SMT instances, involving the set of character strings (**String** symbol), the character string constants and common functions and predicates over strings, like the string length (**str.len** symbol) operator:

```
(set-logic QF_S)
(declare-fun x () String)
(declare-fun y () String)
(assert (not (= x y)))
(assert (= (str.len x) (str.len y)))
```

In this example, the non-standard QF_S logic is composed of the propositional part of first-order logic and of the String Σ -theory.

2.2.3 Complexity of SMT and Practical Applications

Many real-world problems can be encoded as SMT instances [De Moura and Bjørner, 2011]. The principle is similar to encoding a problem as a SAT instance. One needs to express the problem as a set of constraints to be satisfied. If the constraints are written as an SMT instance, an SMT solver can be used to search for a solution to these constraints and in this way solve the problem. Compared to SAT solvers, SMT solvers offer a much richer language for writing the constraints, making it possible to model and solve naturally a much larger set of real-world problems. The ability to use the full first-order syntax and not being limited to propositional formulas makes it possible to write more general constraints. The ability to perform the search modulo some Σ -theories of interest, like integers, arrays or strings, makes it possible to write and solve integer, array or string constraints instead of being limited to boolean constraints only.

Depending on the logic, solving the considered SMT instances can vary from a very simple to a very hard problem. For example, considering decidable logics, linear real programming [Karmarkar, 1984] can be solved in polynomial time, while linear integer programming [Papadimitriou, 1981] is NP-complete. AUFLIA is undecidable in general [El Ghazi and Taghdiri, 2011].

In practice, SMT solvers have been successfully applied to a large variety of real-world problems [Barrett et al., 2009], such as automatic test generation [Cadar et al., 2011], model checking [Henzinger et al., 2003], program verification [Cohen et al., 2009], modeling [Grieskamp et al., 2011], etc.

2.2.4 The DPLL(T) Algorithm for SMT

A Algorithms for SMT

Several solving algorithms have been developed for the SMT problem [Barrett et al., 2009]. However, since the early 2000s, the class of SMT solvers built by extending the DPLL algorithm for SAT have received a growing attention and benefited from many theoretical advances and industrial applications [De Moura and Bjørner, 2011].

The first attempts to exploit the power of SAT solvers to create scalable SMT solvers date back to the 1990s with e.g. [Armando and Giunchiglia, 1993, Bryant et al., 1999, Pnueli et al., 1999]. Historically [Barrett et al., 2009], the very first elements of modern DPLL-based SMT solvers were developed in the late 1970s and in the 1980s, in the context of formal methods like in e.g. [Nelson and Oppen, 1980, Shostak, 1978, Boyer and Moore, 1990]. Following the important advances in SAT in the early 2000s, with the appearance of CDCL-based solvers, modern SMT solvers are based on an extended DPLL engine with CDCL [Barrett et al., 2009, De Moura and Bjørner, 2011] that make use of a well-defined and common scheme for generalizing their DPLL engine to handle non-boolean constraints, called DPLL(T) [Nieuwenhuis et al., 2006]. The principle of DPLL(T) is to integrate the DPLL algorithm with an ad-hoc solver for a Σ -theory, producing in this way an SMT solver for a logic composed of propositional logic and the symbols from the considered Σ -theory. This scheme can be extended to create a solver for a similar logic, but with several Σ -theories at the same time, if one combines the DPLL algorithm with an ad-hoc solver that is not limited to one Σ -theory, but made for a combination of Σ -theories. Such a mixed Σ -theory solver can be created by combining solvers for each of the single Σ -theories, using a common combination approach, called the Nelson-Oppen method [Nelson and Oppen, 1979]. This method works well for most Σ -theory combinations and much research (e.g. [Ghilardi et al., 2005, Tinelli and Zarba, 2005, de Moura and Bjørner, 2008]) has been lead to improve it or extend it to combinations that are not handled. By construction, SMT solvers powered by the DPLL(T) and Nelson-Oppen approaches are limited to logics involving only propositional formulas. However, these solvers can be extended to handle the missing syntactic parts of full first-order syntax. First, by using an ad-hoc Σ -theory solver for uninterpreted functions (considered as a Σ -theory with an empty set of interpreted symbols) [Bachmair et al., 2003]. Secondly, by taking advantage of quantifier handling techniques (e.g. [de Moura and Bjørner, 2008, Reynolds et al., 2013]).

In the remaining part of this section, we detail the main components of DPLL(T)-based SMT solvers. In the next paragraphs, we present the DPLL(T) algorithm. In subsection 2.2.5, we discuss Σ -theory solver combination using the Nelson-Oppen method. In subsection 2.2.6, we review the typical quantifier handling techniques. Subsection 2.2.7 concludes the current section by discussing future challenges for DPLL(T)-based SMT solvers and by briefly describing the two main alternate approaches: direct translation to SAT and use of general purpose first-order theorem provers.

B Ad-hoc Solver for a Σ -theory

The DPLL(T) approach integrates a DPLL solver with an ad-hoc Σ -theory solver to build an SMT solver for propositional satisfiability modulo the Σ -theory. To understand how this integration works, we provide first a formal specification of what an ad-hoc Σ -theory solver is and briefly detail some typical implementations for common Σ -theories.

Definition 2.11. A **solver for a given Σ -theory** is an SMT solver for the logic composed of theories involving only formulas from the considered Σ -theory.

Formally, a first-order formula is called a *formula from a Σ -theory* if it is a conjunction of *literals from this Σ -theory*

$$\begin{aligned} \langle \text{formula} \rangle &::= \langle \text{literal} \rangle && \text{(Literal)} \\ &| \langle \text{formula} \rangle \wedge \langle \text{formula} \rangle && \text{(Logical And)} \\ &| (\langle \text{formula} \rangle) \end{aligned}$$

A first-order predicate (and its negation) is called a literal from a Σ -theory if it only involves either symbols interpreted by the Σ -theory or uninterpreted constant symbols defined over the sets interpreted by the Σ -theory:

$$\begin{aligned} \langle \text{literal} \rangle &::= \langle \text{predicate} \rangle && \text{(Predicate Application)} \\ &| \neg \langle \text{predicate} \rangle && \text{(Negated Predicate Application)} \\ \langle \text{predicate} \rangle &::= \langle \text{symbol} \rangle (\{ \langle \text{term} \rangle , \}^* \langle \text{term} \rangle) && \text{(\Sigma-theory Predicate Application)} \\ &| \langle \text{term} \rangle = \langle \text{term} \rangle && \text{(\Sigma-theory Set Element Equality)} \\ \langle \text{term} \rangle &::= \langle \text{symbol} \rangle (\{ \langle \text{term} \rangle , \}^* \langle \text{term} \rangle) && \text{(\Sigma-theory Function Application)} \\ &| \langle \text{symbol} \rangle && \text{(\Sigma-theory Constant or Uninterpreted Constant over \Sigma-theory set)} \end{aligned}$$

As an example, a Σ -theory solver for the linear integer arithmetic Σ -theory only accepts SMT instances made of formulas that are ANDs of (eventually negated) $\geq$, $>$, $\leq$, $<$ and $=$ predicates over linear integer arithmetic expressions, with their classical interpretation, like, for instance:

$$(x \leq 3) \wedge \neg(y = (z - x)) \wedge (((5 + z) - 3) < z)$$

where x , y , and z are uninterpreted integer constants. Linear integer arithmetic Σ -theory solvers can be based on the Simplex algorithm (e.g. [Dutertre and de Moura, 2006a]).

Similarly, a Σ -theory solver for uninterpreted functions only accepts instances made of formulas that are ANDs of (eventually negated) uninterpreted predicate symbols involving only uninterpreted functions and constant symbols, defined over uninterpreted set symbols. Examples of existing uninterpreted function Σ -theory solvers are those based on congruence closure (e.g. [Bachmair et al., 2003]).

Many Σ -theory solvers have been and are still developed for various Σ -theories, including for arrays (e.g. [Bradley et al., 2006]), strings (e.g. [Hooimeijer and Weimer, 2009, Kiezun et al., 2009, Veanes et al., 2010]), bit-vectors (e.g. [BjØrner and Pichora, 1998]), etc. A detailed discussion over Σ -theory solver design can be found in [Barrett et al., 2009].

C Coupling a SAT and a Σ -Theory Solver to build an SMT Solver

The DPPL(T) [Nieuwenhuis et al., 2006] approach proposes a generic integration scheme between a DPLL solver and a Σ -theory solver. In a nutshell, the SAT solver explores the set of satisfying boolean models for the propositional part of the input SMT instance. The satisfiability modulo the considered Σ -theory of the found boolean models is checked subsequently by the Σ -theory solver. The result is a full SMT solver for propositional satisfiability modulo the Σ -theory. For example, this scheme can be applied to integrate the DPLL algorithm with the Simplex algorithm, producing in this way an SMT solver for the QF_LIA logic.

We demonstrate now, without loss of generality, how the DPPL(T) integration works in general, by using it for solving the following SMT instance:

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(assert (=> (< x 0) (= y 0)))
(assert (or (< x 0) (= y 0)))
(assert (=> (= y 0) (>= x 0)))
```

The first step is to detect all the Σ -theory literals appearing in the instance and to replace them by new boolean constants. This makes it possible to isolate the propositional part of the theory, which is a SAT instance. In this example, there are three linear integer arithmetic literals: $x < 0$ is replaced by the boolean constant a , $y = 0$ is replaced by the boolean constant b and $x \geq 0$ is replaced by the boolean constant c , leading to the following SAT instance:

```
FORMULA (a ==> b)
FORMULA (a v b)
FORMULA (b ==> c)
```

Secondly, the obtained boolean formula is provided to the SAT solver, which searches for a possible satisfying model. If no such a model can be found, this means that the original SMT instance was unsatisfiable and the process is stopped. In our example, satisfying models exist and we suppose that the SAT solver finds the following one:

$$\{a = \text{true}, b = \text{true}, c = \text{true}\}.$$

Thirdly, the Σ -theory solver is asked to find a value for the uninterpreted constants appearing in the Σ -theory literals, so that every literal takes the truth value of its corresponding boolean constant in the model found by the SAT solver. If such values are found, they provide a solution for the considered SMT instance and the process is stopped. In the considered example, the Simplex-based solver must find values for x , y and z , so that:

$$\{(x < 0) \text{ is true}, (y = 0) \text{ is true}, (x \geq 0) \text{ is true}\}.$$

This means to find a satisfying model for the following Σ -theory formula:

$$(x < 0) \wedge (y = 0) \wedge (x \geq 0)$$

which is unsatisfiable because of the first and last literals. As a consequence, the negation of this Σ -theory formula

$$\neg(x < 0) \vee \neg(y = 0) \vee \neg(x \geq 0)$$

is true. The boolean version of this last formula is the clause

$$\neg a \vee \neg b \vee \neg c$$

called a Σ -theory lemma.

Fourthly, the process is repeated recursively, by asking the SAT solver to find another potential solution for the SAT instance, until all the possible solutions to this SAT instance have been tried. If none of these solutions leads to a satisfiable Σ -theory formula, the original SMT instance was unsatisfiable and the process is stopped. At each recursive step, one can prune away the models that have been already tested by adding their corresponding Σ -theory lemma to the theory provided to the SAT solver. For instance, in the considered example, the SAT solver will be asked to find a model for the following SAT instance:

FORMULA $(a \implies b)$

FORMULA $(a \vee b)$

FORMULA $(b \implies c)$

FORMULA $(\neg a \vee \neg b \vee \neg c)$

This instance has a model:

$$\{a = \text{false}, b = \text{true}, c = \text{true}\}.$$

This model leads to ask the Simplex-based solver for a satisfying model for the following Σ -theory formula:

$$\neg(x < 0) \wedge (y = 0) \wedge (x \geq 0)$$

which admits many different models, constituting also models of the original SMT instance. For example, such a model could be

$$\{x = 0, y = 0\}.$$

D Online Integration and Other Improvements

The integration scheme that we have just detailed between a SAT and a Σ -theory solver is called the *offline integration* scheme, as the SAT solver is used as a black-box external component. In most DPLL(T)-based SMT solvers [Barrett et al., 2009], performance gains are typically obtained through a direct *online integration* between the Σ -theory solver and the underlying DPLL algorithm from the SAT solver.

In the offline integration scheme, the Σ -theory solver was called to check the satisfiability modulo the Σ -theory once the SAT solver had found a complete satisfying model for the propositional part of the SMT instance. With the online integration, the Σ -theory solver is called every time DPLL algorithm assigns a value to a boolean constant, checking in this way partial boolean assignments and making an earlier unsatisfiability detection possible.

We demonstrate how this tighter integration works in general, by using it for solving the same SMT instance as in the previous subsection.

The first step is similar to offline integration, where the propositional part of the instance is isolated, leading to the following SAT instance to be handled by the SAT solver:

FORMULA $(a \implies b)$

FORMULA $(a \vee b)$

FORMULA $(b \implies c)$

The SAT solver then transforms these formulas into an equisatisfiable CNF formula like

$$\underbrace{(\neg a \vee b)}_{\text{Clause 1}} \wedge \underbrace{(a \vee b)}_{\text{Clause 2}} \wedge \underbrace{(\neg b \vee c)}_{\text{Clause 3}}$$

on which the DPLL algorithm is executed. Let us suppose that it starts by assigning true to constant c . The Σ -theory solver is then asked to check the satisfiability of this assignment in terms of the underlying literals, i.e. to check for a satisfying model for $x \geq 0$. Such a model exists (for example, $x = 0$), so that the DPLL can continue testing models where c is true. As no consequence can be deduced from c being true, the DPLL algorithm can consider the following formula:

$$\underbrace{(\neg a \vee b)}_{\text{Clause 1}} \wedge \underbrace{(a \vee b)}_{\text{Clause 2}} \wedge \underbrace{\text{true}}_{\text{Clause 3}}$$

and assign for example true to constant a . The Σ -theory solver is asked again to check the satisfiability of the current constant assignment in terms of the underlying literals, i.e. to check for a satisfying model for $(x \geq 0) \wedge (x < 0)$. As this Σ -theory formula is unsatisfiable, the DPLL algorithm can directly use this information to add the learned clause $\neg c \vee \neg a$ and backtrack to assign false to constant a . First, the Σ -theory solver is asked to check the satisfiability of the new boolean assignment $\{c = \text{true}, a = \text{false}\}$, i.e. $(x \geq 0) \wedge \neg(x < 0)$. As it admits many satisfying models like $x = 0$, consequences can be deduced by the DPLL algorithm from the obtained formula:

$$\underbrace{\text{true}}_{\text{Clause 1}} \wedge \underbrace{(\text{false} \vee b)}_{\text{Clause 2}} \wedge \underbrace{\text{true}}_{\text{Clause 3}} \wedge \underbrace{\text{true}}_{\text{Learned Clause 1}}$$

Clause 2 requires true to be assigned to constant b , providing a boolean assignment $\{c = \text{true}, a = \text{false}, b = \text{true}\}$ that make the SAT instance satisfiable. The Σ -theory solver is asked to check the satisfiability of this assignment, i.e. $(x \geq 0) \wedge \neg(x < 0) \wedge (y = 0)$, which admits many satisfying models, like $\{x = 0, y = 0\}$ that are also solution of the general SMT instance.

It should be noted that the online approach is even more efficient if the used Σ -theory solver is incremental and backtrackable [De Moura and Bjørner, 2011], i.e. if new literals can be added and removed from the formula handled by the Σ -theory solver at minimal cost, i.e. without forcing the solver to restart the solving process from scratch.

Finally, many other improvements have been developed to improve the interaction between the DPLL algorithm and the Σ -theory solver. An extensive review of these techniques can be found in [Sebastiani, 2007].

2.2.5 The Nelson-Oppen Method for Theory Combination

In the previous subsection, we have shown how to integrate a DPLL solver with a Σ -theory solver to build a DPLL(T) solver for propositional satisfiability modulo the considered Σ -theory. We have also detailed some existing solver algorithms for common Σ -theories, such as arithmetics or uninterpreted functions. The Nelson-Oppen method [Nelson and Oppen, 1979] is the common technique [Barrett et al., 2009] used in current SMT solvers to extend this integration scheme to logics involving a combination of individual Σ -theories. This method basically enables one to build a solver for the Σ -theory obtained by combining several individual Σ -theories, by integrating the individual solvers for each of the Σ -theories. Such an integrated Σ -theory solver can then be used within the DPLL(T) algorithm to build an SMT solver for propositional satisfiability modulo the combined Σ -theories.

In order to use the Nelson-Oppen method, the underlying Σ -theories must be *disjoint* and *stably-infinite*:

Definition 2.12. Σ -theories are **disjoint** if they do not share any symbol, except from the set equality predicate symbol ($=$).

Definition 2.13. A Σ -theory is **stably-infinite** if, for every satisfiable theory made of formulas from this Σ -theory, at least one of its satisfying models involves an infinite number of possible semantic values.

For example, integer arithmetic is stably infinite because, as the set of integers is infinite, all possible models involve an infinite number of possible semantic values.

Many of the Σ -theories combinations considered in practice fulfill these two conditions [De Moura and Bjørner, 2011]. A lot of research work has been performed, for more than ten years, to generalize the Nelson-Oppen method to non-disjoint and non-stably-infinite Σ -theories. These works involve a deep theoretical analysis which is out of scope of this thesis. The interested reader can find an introductory survey to these works in [Sebastiani, 2007].

More technical details about the Nelson-Oppen method can be found in appendix A.

2.2.6 Handling Quantifiers

A Context

Quantifiers are the root cause why first-order logic is undecidable and all the common SMT-LIB logics allowing to write quantified theories, such as AUFLIA, are not decidable in general. However, in recent years, several procedures have been integrated into DPLL(T)-based SMT solvers, to make them able to find satisfying models or prove the unsatisfiability of some quantified theories. Given a quantified theory, a solver equipped with these procedures can either answer that the theory is (un)satisfiable, if the procedures are successful, or that the satisfiability status is unknown, if the procedures fail.

The DPLL(T) and the Nelson-Oppen techniques have been widely studied for many years and constitute the standard substrate on which all modern SMT solvers are built [Barrett et al., 2009]. In contrast, efforts to develop quantifier handling techniques are recent and achieved concurrently by different solver manufacturers, so that no common framework has still clearly emerged. These efforts typically involve a deep theoretical background, while enlarging the scope of decided quantified theories is a topic of ongoing research. In this subsection, we briefly present the main existing quantifier handling techniques, as used in combination within several modern SMT solvers, and provide references to detailed descriptions of these approaches.

B Quantified Literals

Compared to quantifier-free logics, quantified logics allow to write theories involving quantified literals, to be handled by the DPLL(T) solver. For example, if the following quantified theory:

```
(set-logic LIA)
(declare-const a Int)
(declare-const b Int)
(declare-fun f (Int) Int)
```

```

(assert (> (f a) b))
(assert (=> (= a 1) (forall ((x Int)) (= (f x) b))))
(assert (exists ((x Int)) (forall ((y Int)) (= (f y) (+ x a)))))

```

is passed to the DPLL(T) algorithm, it will detect four literals, including two quantified ones, leading to the following SAT instance:

```

FORMULA  $x$ 
FORMULA  $(y \implies z)$ 
FORMULA  $w$ 

```

where x replaces the literal $f(a) > b$, y replaces the literal $a = 1$, z replaces the quantified literal $\forall x, f(x) = b$ and w replaces the quantified literal $\exists x, \forall y, f(y) = x + a$.

The techniques presented in the next subsections are designed to be integrated within an SMT solver, based on the DPLL(T) and Nelson-Oppen techniques. They make the solver able to find models or prove unsatisfiability in the presence of quantified literals.

C Quantifier Elimination

Quantifier elimination techniques enable one to transform quantified literals into equisatisfiable literals where all or some quantifiers have been eliminated. If the resulting theory is quantifier-free, then the classical DPLL(T) and Nelson-Oppen techniques can be applied. Otherwise, the methods proposed in the next subsections can be used to handle, in some cases, the remaining quantified literals.

Skolemization (see e.g. [Hodges, 1997] for a rather recent discussion of the technique) is the most common quantifier elimination technique. It is applied over every quantified literal as a preprocessing step by many SMT solvers. Basically, it is able to remove all the existential quantifiers appearing in a quantified formula by introducing new uninterpreted constants and functions. If the existential quantifier to eliminate is not part of an universally quantified formula, the quantified variable can be simply replaced by an uninterpreted constant defined over the same set. For example, the following theory:

```

(set-logic LIA)
(assert (exists ((x Int) (y Int) (z Int)) (= (+ x y) z)))

```

is equisatisfiable with

```

(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)
(assert (= (+ x y) z))

```

which can be solved by the classical DPLL(T) algorithm with a LIA Σ -theory solver.

If the existential quantifier to eliminate is part of an universally quantified formula, the existentially quantified variable can be replaced by an uninterpreted function taking the universally quantified variables as parameters and whose image set is the definition set of the replaced variable. For example, the following theory:

```

(set-logic UFNIA)
(assert (forall ((x Int)) (exists ((y Int)) (forall ((z Int)) (= y (* 0 (+ x z)))))))

```

is equisatisfiable with

```
(set-logic UFNIA)
(declare-fun f (Int) Int)
(assert (forall ((x Int)) (forall ((z Int)) (= (f x) (* 0 (+ x z))))))
```

It should be noted that skolemization can only be applied on formulas in *prenex normal form*:

Definition 2.14. *A first-order formula is in **prenex normal form** if no quantifier appears elsewhere than at the very beginning of the formula, i.e. if the formula is written as a list of quantifiers followed by a quantifier-free formula.*

Every first-order formula can be transformed into an equivalent formula in prenex normal form [Hodges, 1997]. Once the quantified literals of a theory properly transformed, skolemization can be applied over each of them to eliminate the existential quantifiers.

Many other quantifier elimination techniques have been developed, typically for formulas belonging to a particular logic (see e.g. [Monniaux, 2010]). As an example, Cooper’s algorithm [Cooper, 1972] can be applied over formulas belonging to the LIA logic.

D Pattern-Based Quantifier Instantiation

Pattern-based quantifier instantiation makes it possible to detect the unsatisfiability of some theories containing universally quantified literals. The principle is to instantiate the universally quantified variables in the quantified formula by constants appearing in the theory, in order to detect a contradiction in the theory. For example, let us consider the following theory:

```
(set-logic UFLIA)
(declare-sort S)
(declare-fun f (S) Int)
(declare-const c S)
(assert (= (f c) 13))
(assert (forall ((s S)) (= (f s) 0)))
```

By instantiating the variable s in the universally quantified literal by the constant c appearing in the theory, one can deduce that the following literal will be true in any model satisfying the theory:

```
(assert (= (f c) 0))
```

By grouping this literal with the other quantifier-free literals in the theory, one obtains the following theory:

```
(set-logic QF_UFLIA)
(declare-sort S)
(declare-fun f (S) Int)
(declare-const c S)
(assert (= (f c) 13))
(assert (= (f c) 0))
```

which can be proven unsatisfiable by a classical DPLL(T) solver, implying that the original theory was also unsatisfiable.

The main challenge in pattern-based quantifier instantiation is to find the ‘good’ constant values for instantiating the quantified variables, which may lead to a contradiction. Typically, this is achieved by associating a pattern to the quantified literal, i.e. a predicate or

term involving all the universally quantified variables as well as, eventually, some constant, function or predicate symbols.

For the example considered above, a pattern for the quantified literal

`(assert (forall ((s S)) (= (f s) 0)))`

could be “(f s)” or “(2 + s)”, as they are both predicates involving the variable *s*.

When the solver meets, within the solved theory, a variable-free predicate or term that matches the pattern, the predicate or term’s constants matching the pattern’s variables are used to instantiate the quantified formula.

In our example, when the solver meets the literal “(assert (= (f c) 13))” in the theory, it detects that the term “(f c)” matches with the pattern “(f s)”. As the constant *c* matches with the variable *s*, it is used to instantiate the variable in the universally quantified literal, enabling the detection of a contradiction, as shown previously.

The pattern to associate with an universally quantified formula can be selected either by the solver, using heuristics, or cleverly crafted by the solver’s user. The SMT-Lib language suggests a (non-standard yet) syntax for associating user-defined patterns to quantified formulas. As an example, considering the theory discussed above and the pattern “(f s)”, we have:

```
(set-logic UFLIA)
(declare-sort S)
(declare-fun f (S) Int)
(declare-const c S)
(assert (= (f c) 13))
(assert (forall ((s S)) (! (= (f s) 0) :pattern ((f s)) )))
```

Given a pattern, the search for matching terms or predicates within the theory is enforced using classical e-matching algorithms (see e.g. [Moskal et al., 2008]).

Originally proposed in the Simplify solver [Detlefs et al., 2005], the pattern-based quantifier instantiation technique has been integrated, with a number of refinements, in modern solvers (see e.g. [Moura and Bjørner, 2007]).

E Integrating DPLL(T) with Inference Rules

One can integrate *inference rules* for first-order logic into the DPLL(T) process, in order to detect the unsatisfiability of some theories involving universally quantified literals.

Definition 2.15. An *inference rule* is a rule establishing that, given a particular formula in a logic, eventually enforcing some properties, a second formula in the logic will be satisfied by any model satisfying the first-one.

An example of an inference rule for first-order logic is

$$\frac{\neg p(a) \wedge \forall s, (p(s) \vee q(s))}{q(a)}$$

where *p* and *q* are predicates and *a* an uninterpreted function. The classical notation is used, stating that the formula on the top implies the formula at the bottom.

Inference rules are very important in theory reasoning and many have been developed.

Definition 2.16. A set of inference rules is *refutational complete* in a logic if, for any unsatisfiable formula in the logic, there exists a finite sequence of rule applications showing that the formula implies the false formula.

Examples of sets of inference rules that are refutational complete for first-order logic are the resolution calculus [Robinson, 1965] and the superposition calculus [Nieuwenhuis and Rubio, 2001]. Given a formula admitting no model in first-order logic, they can thus be used to deduce implied formulas and prove in this way the unsatisfiability of the formula.

In [Bonacina et al., 2009], the DPLL(T) algorithm is coupled with a prover able to apply the inference rules of the superposition calculus. Every time that the DPLL(T) algorithm select a truth value for a literal, the literal is transmitted to the prover. The prover use the inference rules to eventually establish the unsatisfiability of the conjunction of the literals it has received. In parallel, it also uses the inference rules over this conjunction of literals to try to establish the truth value of some still non-assigned literals, potentially enabling the DPPL(T) algorithm to detect a conflict.

As an illustrative example, the following theory can be provided to a solver coupling DPLL(T) with an inference prover:

```
(set-logic UF)
(declare-sort S)
(declare-fun p (S) Bool)
(declare-fun q (S) Bool)
(declare-const a S)
(assert (forall ((s S)) (or (p s) (q s))))
(assert (not (p a)))
(assert (not (q a)))
```

The DPLL(T) will first detect three literals, leading to the following SAT instance:

FORMULA x
FORMULA y
FORMULA z

where x replaces the literal $\forall s, (p(s) \vee q(s))$, y replaces the literal $\neg p(a)$ and z replaces the literal $\neg q(a)$. The SAT instance can be directly transformed into the following single CNF formula:

$$\underbrace{x}_{\text{Clause 1}} \wedge \underbrace{y}_{\text{Clause 2}} \wedge \underbrace{z}_{\text{Clause 3}}$$

Let us suppose that the DPLL(T) algorithm has already assigned true to constants x and y . The inference prover will then be provided with the corresponding literals that must be true, leading to the following conjunction of literals:

$$\neg p(a) \wedge \forall s, (p(s) \vee q(s))$$

In this case, the prover cannot establish that this conjunction is unsatisfiable, which would force the DPLL algorithm to backjump. However, by applying the inference rule

$$\frac{\neg p(a) \wedge \forall s, (p(s) \vee q(s))}{q(a)}$$

the prover can establish that the literal $q(a)$ should be true, i.e. that the boolean constant z will be false in any satisfying model, considering the choices made by the DPLL(T) algorithm so far. The prover will thus transmit the learned clause $\neg z$ to the DPLL algorithm, making it possible to detect a conflict with the third clause. After backjumping, the DPLL(T) algorithm will be able to prove that the original theory was unsatisfiable.

The SMT solver proposed in [Bonacina et al., 2009], integrating DPLL(T) with superposition calculus, is refutational complete for first-order logic. Given a theory in the UF logic which admits no model, it can always prove this theory unsatisfiable. For other logics, where unsatisfiability should be proven modulo some Σ -theories, the solver may remain refutational complete for theories enforcing some particular syntactic restrictions (see [Bonacina et al., 2009] for a discussion). In other cases, the solver may fail to detect properly unsatisfiability when it occurs.

F Model-Based Quantifier Instantiation

If some universal quantifiers remain in the theory once the quantifier elimination techniques applied, the previous approaches only offer the ability, for some unsatisfiable theories, to explicitly establish that they are unsatisfiable. In particular, they do not provide any mean to prove a quantified theory satisfiable and to find a satisfying model for such a theory. Recently, new techniques performing *model-based quantifier instantiation* (MBQI) have brought this missing ability to SMT solvers, at least for some families of quantified theories.

Given a DPLL-provided truth value for each literal in the theory to be solved, MBQI first builds a model that provides the set of quantifier-free literals with their expected value. Then, it derives from this model a candidate model for the whole theory and checks that this candidate model also provides the quantified literals with their expected value. This last step basically means to instantiate completely each quantified literal appearing in the theory, in order to establish its truth value in the context of the candidate model. Given some syntactic restrictions over the input theory and an adequate candidate model builder, this instantiation is guaranteed to be finite. If the candidate model passes the check, the theory is satisfiable and the candidate model is returned as a satisfying one. Otherwise, the conflicting instances of the quantified literals are used to prune out all models that will involve similar conflicts. The process is then repeated. If, at any moment, the quantifier-free part of the theory is unsatisfiable, then the whole theory can be established as unsatisfiable as well.

The interested reader will find a detailed description of the two main existing MBQI approaches in [Ge and Moura, 2009] and [Reynolds et al., 2013]. These two techniques essentially differ in the algorithm used for candidate model building and checking. [Ge and Moura, 2009] uses the classical DPLL and Nelson-Oppen parts of the solver both to build the model for the quantifier-free literals and to check that it provides the quantified literals with their expected values. Considering only decidable Σ -theories, the authors have proven their approach able to solve any SMT instances belonging to the so-called *essentially uninterpreted fragment*. Theories part of this fragment involve quantified variables appearing only as arguments of uninterpreted functions and predicates. The technique, which has been integrated within the Z3 [De Moura and Bjørner, 2008] solver, can also solve some theories outside of this fragment. The concurrent approach proposed in [Reynolds et al., 2013] has been integrated within the CVC4 [Barrett et al., 2011] solver and is able to handle theories where the quantifiers range only over uninterpreted or finite sets. This technique searches for a *finite model* for the quantifier-free literals, i.e. a model which associates every uninterpreted set with a finite set as a semantic value. As a consequence, all the quantifiers can be instantiated completely during the model checking step, as they all range only over finite sets. Methods are provided to build candidate models involving uninterpreted sets of minimal cardinality and to instantiate quantifiers in an efficient way. The technique is claimed to be competitive with [Ge and Moura, 2009].

As a simple illustrative example of the general MBQI process, the following SMT instance can be provided to a solver using the technique proposed in [Ge and Moura, 2009]:

```
(declare-sort S)
(declare-fun f (S) Int)
(declare-const c S)
(assert (>= (f c) 0))
(assert (forall ((s S)) (>= (- (f s) 1) 0) ))
```

One can easily check that this theory belongs to the essentially uninterpreted fragment, as s only appears as an argument of the uninterpreted function f . The DPLL(T) part of the solver will first detect two literals, leading to the following SAT instance:

FORMULA a

FORMULA b

where a replaces the literal $f(c) \geq 0$ and b replaces the literal $\forall s, f(s) - 1 \geq 0$. The SAT instance can be directly transformed into the following single CNF formula:

$$\underbrace{a}_{\text{Clause 1}} \wedge \underbrace{b}_{\text{Clause 2}}$$

The DPLL(T) algorithm will obviously assign true to both constants. The technique from [Ge and Moura, 2009] can then be used to find a satisfying model for the formula where these two literals must be true. First, it isolates the quantifier-free part of this formula:

```
(declare-sort S)
(declare-fun f (S) Int)
(declare-const c S)
(assert (>= (f c) 0))
```

and use the solver's quantifier-free reasoning capabilities to find a satisfying model, like:

$$\begin{aligned} S &= \{c\} \\ f(c) &= 0 \end{aligned}$$

The technique then checks that this candidate model makes the quantified literal $\forall s, f(s) - 1 \geq 0$ true, i.e. that its complete instantiation $f(c) - 1 \geq 0$ is true, where $f(c) = 0$. This means using the traditional solver capabilities to establish whether the following theory is satisfiable:

```
(assert (>= (- 0 1) 0))
```

which will obviously return false. As a consequence, the candidate model is rejected and the conflicting quantifier instance $f(c) - 1 \geq 0$ is added to the the quantifier-free part of the formula, which becomes:

```
(declare-sort S)
(declare-fun f (S) Int)
(declare-const c S)
(assert (>= (f c) 0))
(assert (>= (- (f c) 1) 0) )
```

producing the following candidate model:

$$\begin{aligned} S &= \{c\} \\ f(c) &= 1 \end{aligned}$$

One can check that this model makes the quantified literal $\forall s, f(s) - 1 \geq 0$ true by using the solver capabilities to establish whether the following theory is satisfiable:

`(assert (>= (- 1 1) 0))`

which obviously returns true. The candidate model is thus a satisfying model for the original quantified theory, which is thus satisfiable.

2.2.7 Challenges and Alternatives

Founded upon earlier works, the rise of DPLL-based SMT solvers has occurred in the last decade. These strong advances in solving technologies have had an important impact on a number of application domains [De Moura and Bjørner, 2011], like program testing, model checking, static analysis, software verification as well as software modeling. The other way around, the experimental feedback obtained in this way has raised a set of new theoretical challenges and triggered new advances in SMT solver development [De Moura and Bjørner, 2011], like more effective data structures, more efficient algorithms, tighter integration of solver's component procedures and quantifier reasoning.

It can reasonably be supposed that this process should carry on in the following years, notably as new applications have been emerging [De Moura and Bjørner, 2011], such as runtime analysis of real-time embedded systems or program synthesis. At the same time, many current challenges should remain a strong driving force for research over SMT solvers. As signaled in the previous subsections, the development of efficient procedures for handling new Σ -theories and new Σ -theory combinations is still challenging. This is also true for the development of a faster and wider handling of SMT instances belonging to undecidable logics, notably in presence of quantifiers. Finally, SMT solver scalability in real industrial applications is a matter of discussion in the SMT research community. This should remain an issue in the future, as SMT solvers typically combine algorithms solving NP-Complete problems, which have an exponential worst-case execution time, such as the core DPLL algorithm itself, but also as some other important solver components, like, for example, the linear integer arithmetic Σ -theory solver.

The two main concurrent approaches to DPLL-based SMT solvers are either specializing first-order theorem provers or translating SMT problems into equisatisfiable SAT problems. General-purpose first-order theorem provers are tools designed to establish whether a first-order theory is valid or not, i.e. if this theory is true in any possible model or not. Example of such provers are those based on a recursive application of a set of inference rules, such as the resolution or superposition calculus. Given a theory f , a first-order theorem prover can also be used to establish whether this theory is satisfiable or not. As, logically, a theory f is satisfiable if and only if the theory $\neg f$ is not valid, one can always use a theorem prover to establish the validity of $\neg f$ and thus the satisfiability of f . However, as validity is a semi-decidable and satisfiability an undecidable property in first-order logic, first-order theorem provers may fail to prove that some theories are satisfiable.

If first-order theorem provers are suited to establish general satisfiability in first-order logic, they can also be specialized to establish satisfiability modulo some particular Σ -theories. In order to do so, a set of axiom formulas $a_1, a_2, \dots, a_n$, describing the symbols of the Σ -theory, is necessary. Given such an axiomatization, establishing the satisfiability of a theory t , modulo the Σ -theory, is equivalent to establishing that the theory $a_1 \wedge a_2 \wedge \dots \wedge a_n \wedge t$ is satisfiable, which can be performed by a theorem prover. However, no applicable

axiomatization is available for many common Σ -theories [Barrett et al., 2009], such as, for example, real arithmetics (which require an infinite number of axioms) and integer arithmetic (which cannot be described by any decidable set of axioms). Moreover, the performance of theorem provers is described as “often unacceptable” [Barrett et al., 2009]. Several works have investigated an integration between theorem provers and DPLL-based SMT solvers. We have previously discussed of [Bonacina et al., 2009], which integrates inference rules from theorem provers into DPLL(T) to improve quantifier handling. Some other works, like [Korovin and Voronkov, 2007, Althaus et al., 2009], have integrated an LIA Σ -theory prover into a theorem prover.

DPLL(T)-based tools expands an SMT solver out of a SAT solver. This is called the *lazy* approach to SAT-based SMT solving. Other tools, like UCLID [Lahiri and Seshia, 2004], use a different approach, called the *eager* approach, by compiling SMT problems into equisatisfiable SAT problems, which they provide to an off-the-shelf SAT solver. Such an exact translation is only possible for decidable quantifier-free theories. In practice, the translation process typically combines Σ -theory specific compiling rules. It also involves optimization steps to try to avoid, when possible, the generation of large SAT instances, which may be intractable by current SAT solvers [Barrett et al., 2009].

2.3 Relational Solvers

2.3.1 Writing and Solving Relational Constraints

SAT and SMT solvers, studied in the two previous sections, are useful as they are constraint solvers. Given a problem modeled as a set of constraints to satisfy, they offer a generic language to express the constraints and algorithms to find satisfying models, i.e. values for the constrained objects that fulfill the constraints, if they exist. Propositional logic is the constraint language proposed by SAT solvers. SMT solvers make this constraint language richer, allowing variable, function and predicate symbols, non-boolean constants and quantifiers, as well as symbols with a fixed standard interpretation, defined in so-called Σ -theories. This enrichment of the constraint language comes with a much more complex solving process. State-of-the-art SMT solvers are basically built around a SAT solver expanded to handle the non-propositional part of the SMT language.

The research work detailed in this thesis proposes a constraint-based approach to relational database application testing. It requires to express and solve *relational* constraints, where the constrained objects are relations. The problem of writing and solving such constraints has been already studied in the context of the Alloy framework [Jackson, 2006], developed at MIT. Alloy is a software modeling and verification framework, where the modeled system is represented by relational constraints and verified by solving these constraints. It has been used in many different application domains [Torlak et al., 2013] and the Alloy website¹ references at least one hundred Alloy-based case-study papers. In the first part of this section, we present the Alloy relational constraint language and solver, which form the core part of the framework. The language is based both on first-order logic and relational operators. The solver translates the relational constraints into a SAT instance, which is equivalent modulo a maximum bound over the relation size. An off-the-shelf SAT solver is then leveraged to solve the generated instance. This solving process, if sufficient for many Alloy applications, may come short for others, as it neither makes it possible to search for relations of unbounded size, nor to establish that some relational constraints are unsatisfiable. To overcome these limitations, a recent work [El Ghazi and Taghdiri, 2011] has proposed a different solving process, where the Alloy constraints are compiled into SMT-Lib and solved by the Z3 SMT solver. This work is presented in the second part of this section.

2.3.2 The Alloy Framework

A The Alloy Language

In a nutshell, compared to SAT and SMT, the Alloy constraint language enables using full first-order logic, modulo a Σ -theory of integers and relations. A constraint set expressed in Alloy is typically composed of four main parts: a number of set and relation symbol definitions, a number of constraint definitions over these symbols, an assertion definition and a solving statement. The constraints are expressed by mixing first-order logic with relational predicates and operators. Integer constraints can also be expressed, notably on set and relation cardinalities. The language enables defining macro functions and predicates to be used within the constraints. A particular constraint can be defined as an assertion. The solving statement instructs the solver to search for a model which violates the assertion and satisfies the other constraints. In the context of model-checking, the

¹<http://alloy.mit.edu>

constraints define the valid states of a software system specification, while the assertion defines a property that should hold in every of these system states. The solving statement then enables one to search for a potential violation of this property among the specification. As the search is performed modulo a given bound for the size of the defined top-level sets, the solving statement can be used to define a non-default bound for some or all the considered sets.

We provide now an illustrative example of the Alloy syntax. The interested reader will find a full description of Alloy's syntax and semantics in [Jackson, 2006].

B Example

As an illustrative example of the Alloy language, we detail, in this subsection, a set of Alloy constraints written in the context of modeling the bookshop described at section 1.1.2.

A set of Alloy constraints must always be given a name, prefixed by the **module** keyword:

```
module bookShop
```

Set symbols are defined using the **sig** keyword. For the five classical sets of the bookshop example, as defined at section 1.1.2, we have:

```
sig Identifier {}
sig Label {}
sig Barcode {}
sig Title {}
sig Author {}
```

Symbol definition for sets which are subsets of others use the **in** keyword. For example, the set of belgian authors:

```
sig BelgianAuthor in Author {}
```

Symbols representing an element of a particular set are defined as singleton subsets, using the **one** keyword. For example, the title of the last book copy made available at the bookshop:

```
one sig LastBookTitle in Title {}
```

Alloy provides the built-in set **Int** for integers, used for example here to define an integer constant symbol newIdentifier :

```
one sig newIdentifier in Int {}
```

Relation symbols can notably be defined using an object-oriented syntax. For example, the shelf relation from the bookshop example can be represented in Alloy by:

```
sig Shelf {
  id: one Identifier ,
  lab: one Label
}
```

which can be seen either as a class definition (one class Shelf with two fields id of type Identifier and lab of type Label), or as the definition of the Shelf set and of two binary relations (id defined over Shelf $\times$ Identifier and lab defined over Shelf $\times$ Label). Multiplicity symbols can be used in relation definitions. For example, the **one** symbol used in the id relation definition constrains each element of Shelf appearing in the relation id to be associated to exactly one element of Identifier, i.e. it constrains the relation id to be a

function. N-ary relations can also be defined. For example, an alternative Alloy definition for the shelf relation from the bookshop example could be:

```
sig shelf {
sh: Identifier → Label
}
```

which defines the ternary relation sh over $\text{Shelf} \times \text{Identifier} \times \text{Label}$. It should be noted that this definition is not totally equivalent to the previous one. The relation sh indeed represents an arbitrary relation, with no functional constraint, contrary to the id and lab relations.

One can write constraints over the defined element, set and relation symbols using the **fact** keyword. For example, the following constraint requires the Title set to be the empty set (represented by the keyword **none**):

```
fact {Title=none}
```

Integer constraints can be written as well, notably over the cardinality (#) of the defined sets:

```
fact {newIdentifier = #Identifier.add[1]} // add function is integer +
```

Alloy offers three essentially equivalent constraint languages: first-order logic, a set of relational predicates and operators, as well as a “navigational” language. For example, considering the following set symbol, representing all the shelves in the bookshop at time t:

```
sig ShelfAtTimeT in Shelf {}
```

The constraint saying that “there cannot be two shelves in the bookshop sharing a single identifier at time t” can be expressed in first-order logic as

```
fact {
all s1,s2: ShelfAtTimeT, id1,id2: Identifier |
((s1→id1 in id) and (s2→id2 in id) and id1 = id2) implies s1 = s2
}
```

which is the Alloy equivalent of

$$\forall s1 \in \text{ShelfAtTimeT}, \forall s2 \in \text{ShelfAtTimeT}, \forall id1 \in \text{Identifier}, \forall id2 \in \text{Identifier}, \\ ((s1, id1) \in id \wedge (s2, id2) \in id \wedge id1 = id2) \implies s1 = s2$$

Using the relational style, the same constraint can be expressed as:

```
fact {
no (ShelfAtTimeT <:id).~(ShelfAtTimeT <:id) – iden
}
```

The apparently complex syntax of this constraint can be explained by evaluating its value over the following content for the ShelfAtTimeT set and the id relation:

```
ShelfAtTimeT = {S1, S2, S3}
id = {(S1, id1), (S2, id1), (S3, id2), (S4, id3)}
```

The <: relational operator *reduces the range* of the relation on the right to be equal to the set on the left, by “deleting” the necessary elements:

$$\text{id} \prec \text{ShelfAtTimeT} = \{(S1, id1), (S2, id1), (S3, id2)\}$$

The $\sim$ relational operator *transposes* the argument relation, by “inverting” all its elements:

$$\sim(\text{id} \prec \text{ShelfAtTimeT}) = \{(id1, S1), (id1, S2), (id2, S3)\}$$

The $\cdot$ relational operator *composes* two relations to build a new relation. First, it joins any tuple from the left relation to any tuple of the right relation, provided that the last element of the left tuple matches the first element of the right tuple. Secondly, it drops the matching elements in the resulting tuple:

$$\begin{aligned} & (\text{ShelfAtTimeT} \prec \text{id}) \cdot \sim(\text{ShelfAtTimeT} \prec \text{id}) \\ &= \{(S1, S1), (S1, S2), (S2, S1), (S2, S2), (S3, S3)\} \end{aligned}$$

The $-$ relational operator is the set difference and the **iden** keyword represents the identity function over the set of all the possible set elements. We have thus:

$$\begin{aligned} & (\text{ShelfAtTimeT} \prec \text{id}) \cdot \sim(\text{ShelfAtTimeT} \prec \text{id}) - \mathbf{iden} \\ &= \{(S1, S2), (S2, S1)\} \end{aligned}$$

The **no** predicate requires the argument set to be empty. As a consequence, we have that the constraint:

```
fact {
no (ShelfAtTimeT <id) · ~ (ShelfAtTimeT <id) - iden
}
```

is false for the considered set and relation contents, which is correct, as the two shelves S1 and S2 share the same identifier. One can check that the constraint will be true every time that all the shelves present at time t in the bookshop have unique identifiers.

Using the navigation style, the same constraint can be expressed as:

```
fact {
all i: Identifier | lone (id · i & ShelfAtTimeT)
}
```

where $\&$ is the set intersection symbol, stating that for each identifier i, the set of shelves at time t which are identified by i contains no more than one element. In a nutshell, the navigational style makes it possible to mix the logic and relational styles to offer a compromise between verbosity and complexity. It is the recommended style to write Alloy constraints.

Macro functions and predicates can be defined to make the constraints more readable. For example, the following constraint over the bookshop’s temperature makes use of the helper predicate `isExtreme` and function `abs`:

```
one sig temperatureInBookshop in Int {}
fact { not isExtreme[temperatureInBookshop] }
```

```
pred isExtreme[temp: one Int]
{abs[temp]> 40}
fun abs[i: one Int] : one Int
{i < 0 => 0-i else i}
```

An assertion is defined as a named constraint, prefixed by the **assert** keyword. A solving statement is composed of the **check** keyword followed by the assertion name. For example:

```
one sig numberOfBookCopiesInBookshop in Int {}
assert bookShopNeverEmpty{numberOfBookCopiesInBookshop > 0}
check bookShopNeverEmpty
```

will instruct the solver to search for a model for the defined set and relation symbols, enforcing the defined constraints, and violating the assertion, stating that the bookshop should at least contain one book copy.

The Alloy solver performs a SAT-based exhaustive model search by bounding the maximal cardinality of the defined top-level sets. By default, the solver will search for models where each top-level set contains at most three elements. The **check** command enables one to modify this default behavior. For example, the following statement sets the bound to 5 for all sets, except for the Identifier set, for which the bound remains 3:

```
check bookShopNeverEmpty for 5 but 3 Identifier
```

Handling the cardinality of the built-in integer set is different, because the Alloy solver encodes integers as bit vectors. For example, the following statement:

```
check bookShopNeverEmpty for 4 Int
```

instructs the solver to consider integers coded over 4 bits, i.e. integers from -8 up to 7.

C The Alloy Solver

The Alloy framework [Jackson, 2006] comes with an ad-hoc solver for constraints written in the Alloy language, called the Alloy Analyzer. As the Alloy language is based on full first-order logic, solving Alloy constraints is undecidable in general. The solution adopted in the Alloy Analyzer is to perform an exhaustive SAT-based search, among the finite number of models enforcing maximal cardinality bound for the defined top-level sets. The solver first compiles the constraints into a core Alloy syntax [Torlak and Jackson, 2007]. The resulting core constraints are subsequently translated into a SAT instance, using a standard boolean representation for cardinality-bounded sets and relations as well as bit-blasting for integers (see e.g. [Bryant et al., 2009] for a discussion over bit-blasting). The original constraints and the SAT instance are equivalent, if one considers only the models bounded by the maximal set cardinalities. The SAT instance is then solved using an off-the-shelf SAT solver, like Minisat [Eén and Sörensson, 2003] or SAT4J [Berre and Parrain, 2010], which can be chosen by the user among a list. If a satisfying model is found for the SAT instance, it is translated back into constraint-satisfying values for the set and relation symbols in the original problem. If the SAT instance is unsatisfiable, it does not mean that the original Alloy constraints were unsatisfiable. It only means that no solution exists considering the defined set cardinality bounds, as one could potentially exist if the bounds were increased. Skolemization and several optimizations, including symmetry-breaking [Torlak and Jackson, 2007], are performed during the SAT-translation process.

The principle and limits of the SAT-translation process used by the Alloy analyzer for relational constraints can be illustrated over the following constraints²:

²These constraints define two top-level set symbols 'Umbrella' and 'Person' and a relation symbol 'has' representing an arbitrary relation over $Umbrella \times Person$. It defines an assertion saying that at least

```
module rainWorld
```

```
sig Umbrella { }
```

```
sig Person { has: set Umbrella }
```

```
assert atLeastSomeoneHasAnUmbrella{some s:Person | some s.has }
```

```
check atLeastSomeoneHasAnUmbrella for exactly 2 Person, exactly 2 Umbrella
```

The solver is instructed to search for models where the two top-level sets Umbrella and Person contain exactly two elements. We have thus:

$$\text{Umbrella} = \{Umbrella0, Umbrella1\}$$

$$\text{Umbrella} = \{Person0, Person1\}$$

As a consequence, the solver needs only to establish which tuples will be present in the has relation, among the four possible ones: $(Person0, Umbrella0)$, $(Person0, Umbrella1)$, $(Person1, Umbrella0)$, $(Person1, Umbrella1)$. If each tuple is associated with a boolean constant, flagging its presence in the relation, the negation of the Alloy assertion is equivalent with the following SAT instance:

$$\neg(b_{00} \vee b_{01} \vee b_{10} \vee b_{11})$$

which admits a single solution where the four flags are false. As a consequence, the solver has found a satisfying model: the one where the has relation is empty. In such a model, nobody has an umbrella and the assertion is violated.

The process described in this example can be easily generalized to the case where the top-level set cardinalities are bounded by a maximal value and no more fixed to a precise value. One can notice that, considering more complex models, the size of the generated SAT instance may explode as the maximal set cardinality value is increased. This SAT instance may thus become intractable even for modern solvers. The Alloy authors [Jackson, 2006] have claimed the so-called “small scope hypothesis”, saying that, in most practical cases, if an assertion can be violated, then a small model that violates it should exist, so that searching for models with a small cardinality bound will be often meaningful.

2.3.3 From Alloy to SMT

As the Alloy Analyzer performs a bounded search, it cannot establish the unsatisfiability of constraints, and thus cannot prove that some assertions are verified in all the possible states of a modeled system. Several works have tried to provide this ability to the Alloy framework. In order to do so, [Arkoudas et al., 2004, Frias et al., 2007] use theorem provers and [El Ghazi and Taghdiri, 2010, El Ghazi and Taghdiri, 2011] leverage an SMT solver to solve Alloy constraints. This last approach is particularly relevant for this research work, as SMT solvers are at the core of the constraint-based testing paradigm, used in this thesis. We describe thus this SMT-based approach in the remaining part of this subsection.

[El Ghazi and Taghdiri, 2011] proposes to modify the Alloy constraint solving process by translating the constraints into an SMT instance instead of a SAT instance. As SMT can cover full first-order logic and integer arithmetics, the translation process does not

one person has at least one umbrella. Finally, it instructs the solver to search for a model violating this assertion, considering only models where the cardinality of the Umbrella and Person sets is two.

require set finitization, and the SMT solver can perform an unbounded model search. As a consequence, if the SMT instance is proven unsatisfiable, then the original Alloy constraints are unsatisfiable too. The paper describes a set of sound translation rules between the core Alloy language used by the Alloy analyzer and the SMT-Lib language. The output SMT instance fits into the AUFLIA logic and makes an arbitrary use of quantifiers, which makes the problem of solving the generated constraints undecidable in general. This is coherent with the Alloy logic being undecidable. The translated constraints are then solved by the Z3 solver [De Moura and Bjørner, 2008], which may thus answer unknown. The translation rules have been optimized to produce constraints on which the solver performs well in practice, notably considering its quantifier handling capabilities. The technique was evaluated over eight Alloy problems, showing the approach able to detect constraint unsatisfiability almost instantly. If the technique enabled an unbounded model search, it exhibited poor results for satisfying model finding, compared to the Alloy Analyzer. However, the experiments were realized with version 2.16 of Z3, i.e. without the model-based quantifier instantiation capabilities. MBQI, supposed to bring strongly improved model finding capabilities for quantified constraints, was introduced in Z3 2.17, according to the release notes on the Z3 repository³.

³<http://z3.codeplex.com>

Chapter 3

Constraint-Based Software Testing

Contents

3.1	Software Testing	83
3.1.1	Principles and Automation	83
3.1.2	Testing Target and Objective	83
3.1.3	Alternative Approaches	84
3.2	The Constraint-Based Approach	85
3.2.1	Purpose and Principle	85
3.2.2	Symbolic Execution	86
3.2.3	Static and Dynamic Path Exploration	88
3.2.4	Recent Trends	92
3.2.5	Alternative Automatic Test Input Generation Techniques . . .	95

DEFECTS in application code have been found to be the cause of a number of important and well-documented software failures, causing serious damage for the society. One of the most famous example is the Ariane 5 test flight disaster [Dowson, 1997].

Testing remains, in current software engineering practice, the primary approach to detect defects in software. As software testing is a complex and costly activity, automating all aspects of the testing process has raised a strong attention and is at the center of an important amount of research work.

Of particular interest is the automation of *test input generation* for functional unit testing, where the idea is to automatically generate a representative set of inputs for testing a unit of code. *Constraint-based testing* is a particularly successful approach for automating test input generation, where the application code is transformed into SMT instances whose solutions are representative test inputs.

In this thesis, we study how the constraint-based testing state of the art can be extended to automate testing even when the tested code contains SQL statements, enabling an interaction with a relational database.

In this chapter, we introduce the main concepts and issues of software testing (section 3.1) and detail the current state of art in constraint-based testing (section 3.2).

3.1 Software Testing

3.1.1 Principles and Automation

Software testing [Kaner et al., 1993, Jorgensen, 2008] means executing a piece of software on a set of *test cases* (called a *test set*) and verifying that the software behavior matches its expected behavior for each test case. In current software development practice, testing remains the primary approach to discover software defects. Testing is used both to improve software quality, by correcting the detected defects, and to provide a (partial) level of confidence in the quality of the tested software (in so called acceptance testing), by showing that a given test set has not revealed any defect.

Conceptually, testing a piece of software essentially necessitates to solve two orthogonal problems: the *test case selection* problem and the *test oracle* problem. For most software systems, the set of possible test cases is typically infinite or very large, so that exhaustive testing is not possible in practice. Given a testing budget, i.e. the maximal number of test cases that can be verified in practice, the test case selection problem means choosing a test set for locating the defects present in the considered piece of software. Once a test set chosen, the piece of software can be executed on each of the selected test cases. The oracle problem means determining, for a given test case, what the expected software behavior is, and verifying that it matches the actual behavior.

As testing is a complex and repetitive task and as its cost is often important, automated test case selectors and oracles have been advocated as interesting means to improve testing efficiency and to reduce software development costs. The impact of such an automation must be mitigated [Ramler and Wolfmaier, 2006], as automatic testing may also bring development risks, compared to manual testing. However, the potential high benefits of automation have motivated much research to develop relevant automated techniques, to test efficiently all aspects of software.

3.1.2 Testing Target and Objective

Testing a piece of software can cover a large spectrum of different activities, from assessing a small Java method with regard to its functional specification, up to testing the performance and scalability of a whole DMBS in several scenarios. The different testing activities are characterized by their testing target and objective [Abran et al., 2001]. The target defines which part of the software is the object of testing. The objective defines with respect to which desirable software property defects should be detected.

Testing activities are often classified into *unit*, *integration* or *system testing*, according to the scope of the testing target. In unit testing, the target is a single software unit that has been developed and can be tested in isolation, like an individual function or a larger but cohesive component. In integration testing, the target is composed of several software components, developed and previously tested independently, which are interacting together inside the whole system. In system testing, the target is the whole software system, running within its external environment.

An objective of testing can be verifying that the software functional specification is correctly implemented. Testing activities following this objective are referred to as *functional testing*. Other testing activities can try to detect defects with regard to other software properties, such as in *performance*, *security*, *reliability*, *usability*, *user acceptance testing*, etc.

3.1.3 Alternative Approaches

Software testing means finding defects from an analysis of software dynamic behavior. Alternative approaches usually try to locate defects from a direct (static) inspection of the assessed piece of software (or of models of it). One can try to detect manually or automatically common patterns of bugs and bad practices (see e.g. [Mantyla et al., 2003]). The software system can be manually reviewed by (experimented) developers (see e.g. [Da Cunha and Greathead, 2007]). Automated formal methods, like static analysis and model checking, have also been used for defect detection and we briefly describe their principles in the remaining part of this subsection. A detailed review of these techniques can be found in [D’silva et al., 2008].

With static analysis, the source code is analyzed algorithmically to detect the presence of some specific defects in the code (possible division by zero, etc.). For many relevant analysis goals, the problem is undecidable in general, so that the computed result is often approximate, potentially missing some defects and signaling defects where there is none. Model checking is based on the development of a formal model of the software system execution, like a state machine, representing (some of) the possible states of the system. This model is exhaustively examined by algorithms against the violation of some desirable properties. The properties are typically the absence of reachable “error states”, like states violating some assertion in the system code, or the unreachability of “trap states”, leading the system into a blocking situation.

3.2 The Constraint-Based Approach

3.2.1 Purpose and Principle

Constraint-based testing (CBT) [Pasareanu and Visser, 2009, Cadar et al., 2011, Cadar and Sen, 2013] is an approach for the automation of test case selection in the context of functional unit testing. Concretely, given a unit of typically imperative code, like a C function, CBT intends to select automatically the test cases which are the most effective for finding the defects possibly present in this code. Considering a deterministic and non-concurrent code, a test case for such a unit of code is a concrete value for each of the code inputs, like function input parameters, I/O reads, etc. CBT selects automatically such test cases by trying, as detailed in the next paragraphs, to generate groups of inputs that will *trigger* the different *execution paths* from the *control flow graph* of the tested code.

Given a unit of code, its control flow graph [Zhu et al., 1997] is a representation as a directed graph of all the possible ways of executing this code. Each sequence of statements in the code that are always executed linearly is represented by a particular node in the graph. Edges enable modeling the control dependence points, such as conditionals and loops. An edge from a node A to a node B labeled by a predicate P represents the fact that the statements of B are executed after the statements of A if and only if P is true after the execution of the last statement of A. Finally, a begin node and an end node are used to represent the starting and termination points of the computation. As an example, Figure 3.1 presents a sample unit of code with its corresponding control-flow graph.

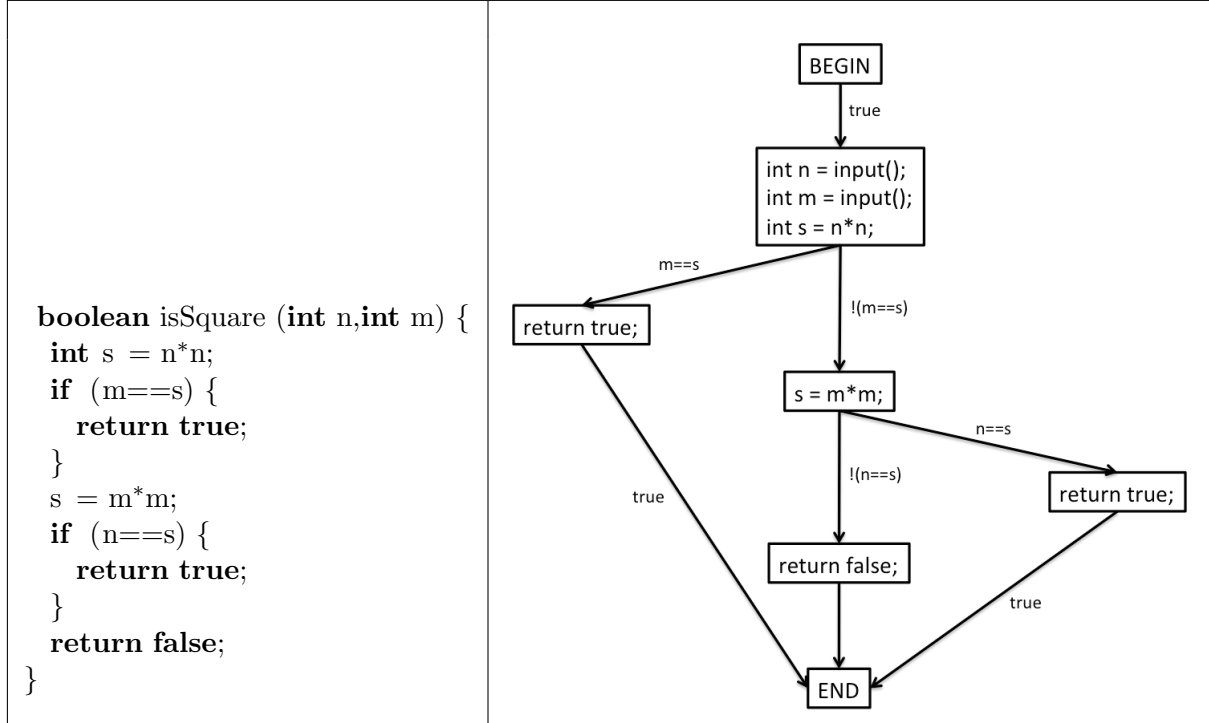


Figure 3.1: Sample unit of code and corresponding control flow graph

Definition 3.1. An *execution path* in a unit of code is a path in its control flow graph going from the begin node to the end node.

Each execution path represents one of the possible behaviors for the unit of code. In this work, we will only consider finite execution paths (i.e. in presence of loops in the code, we will only consider paths executing each of the loops a finite number of time). In the example of figure 3.1, the code has three execution paths, where the first, second and third return statements are respectively executed.

Definition 3.2. A test case *triggers* an execution path if executing the code with the inputs from the test case drives the code execution along this path.

As the code is supposed to be deterministic and non-concurrent, a test case always triggers a single execution path. In our example, the test case $\{n=2, m=4\}$ triggers the first execution path, the test case $\{n=4, m=2\}$ triggers the second execution path and the test case $\{n=2, m=3\}$ triggers the third execution path.

Given an execution path in a unit of code, *symbolic execution* is a process that builds an SMT instance whose each satisfying model describes a test case triggering the path. Symbolic execution is the core mechanism of constraint-based testing: CBT explores, either statically or dynamically, the execution paths of the tested code and uses symbolic execution, coupled with an SMT solver, to generate test cases triggering each of the explored paths. Any of the test cases in the generated test set enables thus the tester to verify a new possible behavior for the tested code, improving the chance to locate a defect still undiscovered in the code. When the code evolves, the test set built for the original code can be used for regression testing of the modified code, checking that it remains at least at the same level of correctness as the original code.

Symbolic execution is detailed in the next subsection. The static and dynamic path exploration processes used in constraint-based testing are discussed in subsection 3.2.3. The current research trends and challenges for CBT are discussed in subsection 3.2.4. The common alternate approaches to CBT are briefly presented in subsection 3.2.5.

3.2.2 Symbolic Execution

The principle of symbolic execution [King, 1976] is to traverse the control-flow graph of the tested unit by executing the code over symbolic inputs instead of concrete ones. Each time a control dependence point is encountered, symbolic execution proceeds along one of the possible paths, generating constraints upon the symbolic inputs such that when the program receives concrete input values that satisfy these constraints, the real execution would proceed along the selected path. When a termination point is reached, the constraints that have been generated along are regrouped in a so-called *path constraint*. This path constraint is solved, resulting in a set of concrete values for the application inputs that make the real execution proceed along the given path. These values describe a test case triggering the considered path.

For example, let us demonstrate symbolic execution over the path where the third **return** statement is executed, in the sample code unit of figure 3.1. First, the method receives input values for the parameters n and m . Symbolic execution initializes these two parameters with the symbolic values n_0 and m_0 . The state of the memory M is thus

$$M \equiv \{n \mapsto n_0, m \mapsto m_0\}$$

Then, the statement **int** $s = n * n$; is executed and the memory becomes

$$M \equiv \{n \mapsto n_0, m \mapsto m_0, s \mapsto n_0 * n_0\}$$

As a first control dependence point is encountered, with the conditional statement, symbolic execution initializes the path constraint PC with a first constraint over the symbolic values. This constraint states that the **if** condition $m==s$ is false, so that the considered path is taken. Replacing the variable m and s by their value in the memory, we have:

$$PC \equiv \{\neg(m_0 = n_0 * n_0)\}$$

As the statement **int** $s = m * m$; is executed, the memory becomes:

$$M \equiv \{n \mapsto n_0, m \mapsto m_0, s \mapsto m_0 * m_0\}$$

Meeting the second **if** statement, the path constraint is updated with a constraint stating that $n==s$ is false:

$$PC \equiv \{\neg(m_0 = n_0 * n_0), \neg(n_0 = m_0 * m_0)\}$$

Finally, as the return statement is executed, a termination point is reached and the path constraint can be solved.

In symbolic execution, path constraints are, by construction, sets of quantifier-free logical formulas mixing constraints over typed objects. They can thus be naturally expressed as SMT instances and solved by SMT solvers. In the considered example, the path constraint $\{\neg(m_0 = n_0 * n_0), \neg(n_0 = m_0 * m_0)\}$ can be expressed in SMT-Lib as:

```
(declare-const m0 Int)
(declare-const n0 Int)
(assert (not (= m0 (* n0 n0))))
(assert (not (= n0 (* m0 m0))))
```

A satisfying model for this instance is $\{n_0 = 2, m_0 = 3\}$, which defines a value for the n and m input parameters, so that the execution takes the path where the third return statement is executed. These values describe thus a test case triggering this path.

The logic in which falls the SMT instance generated by symbolic execution depends on the datatypes and datatype operators used in the executed code. In some cases, the target logic may be undecidable. This makes symbolic execution unable, in general, to find test cases triggering any path in any code. In the previous example, the target logic is composed of quantifier-free theories modulo non-linear integer arithmetic (QF_NIA), which is undecidable in general. However, the simple SMT instance generated in the previous paragraph is perfectly solved by modern SMT solvers.

In some cases, the SMT instance generated by symbolic execution may be unsatisfiable. This means that the path for which the instance was generated is infeasible.

Definition 3.3. *An execution path is **feasible** if there exists at least one set of inputs for the code that triggers the execution of this path.*

For example, in the following code:

```
boolean unfeasible (int n) {
  if (n==0) {
    n = 1;
  }
}
```

```

if (n==0) {
  return true;
}
return false;
}

```

The SMT instance produced by symbolic execution for the path where the body of both if statements is executed is:

```

(declare-const n0 Int)
(assert (= n0 0))
(assert (= 1 0))

```

which is obviously unsatisfiable, because of the second formula. This means that the associated path is unfeasible, which is true as the variable *a* will be 1 when going out of the first if and it should be 0 to enter the second one.

The percentage of paths that are infeasible in some code can be not negligible in practice [Cadaru et al., 2006].

3.2.3 Static and Dynamic Path Exploration

With symbolic execution, constraint-based testing has a mean to generate a test case for a path in a code to test. The way, either static or dynamic, in which the different execution paths are explored for subsequent test case generation is discussed in this section.

In its original [King, 1976] and most simple flavor, constraint based testing use a *static path exploration* scheme. This process explores the unit of code statically, reading it statement after statement, starting by the first one. The traversed code is symbolically executed in parallel. Every time the process meets a control dependence point, the path constraint is updated with a constraint that drives the execution towards one of the possible directions. The resulting partial path constraint is then solved. If this partial path constraint is satisfiable, the process proceeds along the direction for which the path constraint was updated. If the partial path constraint is unsatisfiable, another possible direction is considered and the process is repeated. When the process reaches a computation termination point, the solution of the last path constraint solved constitute a test case, triggering the full path that has just been explored. The process then backtracks to the last control dependence point met and proceeds along another direction. When all the possible directions have been tried for a given control dependence point, the process backtracks to the last met control dependence point, if it exists. Otherwise, it stops, as all the possible paths have been explored.

For example, considering the following unit of code:

```

boolean foo (int n) {
  int m = 0;
  if (n > 0 | m == 0)
    return true;
  else
    return false; }

```

The process starts and reads the first statement **int** m = 0. In parallel, it performs symbolic execution and the memory becomes:

$$M \equiv \{n \mapsto n_0, m \mapsto 0\}$$

The next statement is an **if** control dependence point. The process tries a first direction, where the then branch is taken. The corresponding path constraint is

$$PC \equiv \{n_0 > 0 \vee 0 = 0\}$$

for which $n_0 = 1$ is a possible satisfiable model. As the path constraint is satisfiable, the process proceeds along the considered direction and meets the **return true**; statement which is a computation termination point. As a consequence, the solution of the last path constraint solved ($n_0 = 1$) constitute a test case, triggering the full path that has just been explored (the one which ends with the **return true**; statement). The process then backtracks to the last met control dependence point (the **if** statement) and tries the only other possible direction: the one where the else branch is taken. The path constraint becomes

$$PC \equiv \{\neg(n_0 > 0 \vee 0 = 0)\}$$

which is unsatisfiable as $0 = 0$ is true. As all the possible directions have been explored for the **if** statement, the process then tries to backtrack to the last met control dependence point. As there is no, the process stops as all paths have been analyzed.

Conceptually, the static path exploration scheme performs a depth first traversal of the control-flow graph, pruning away the unfeasible branches as soon as they are met. In presence of loops in the control-flow graph, the search is bounded to a maximal depth [Cadaru and Sen, 2013], to guarantee the termination of the process.

Originally proposed in 1976, constraint-based testing has gained an renewed attention in the last decade. This interest is said to have two main causes. First, the emergence of modern DPLL(T)-based SMT solvers, coupled with the availability of an increased computer power, have provided a practical computational basis for solving the constraints generated by symbolic execution. Secondly, new techniques combining symbolic execution with a *dynamic path exploration* scheme [Godefroid et al., 2005, Sen et al., 2005, Cadaru et al., 2006] have made possible to test code even in presence of the (frequent) statements that cannot be symbolically executed, such as proprietary API calls, thanks to the direct availability of runtime values for these statements, bringing a new confidence in the success of CBT.

Dynamic path exploration starts by executing the unit of code on randomly or heuristically chosen concrete inputs. The code is instrumented so that symbolic execution is performed together with this normal run of the program, thereby generating the path constraint corresponding to the concrete execution. By flipping the constraint corresponding to the last control dependence point met in the concrete execution, one produces a path constraint whose solution describes a new test case, that will trigger the execution of a different path. The unit of code is then executed on these new inputs and the process is repeated. If each of the possible directions for a given control dependence point has either been explored or has generated an unsatisfiable path constraint, the process backtracks to the control dependence point that has been met previously, if it exists. Otherwise, it stops, as all the possible paths have been explored.

For example, considering the following unit of code:

```
int bar (int n) {
  if (n <= 0)
    return 1;
  else
```

```

    return 3;
}

```

The code is first instrumented in some way to enable symbolic execution together with the normal program execution. For example, the following code could be added to the original code:

```

int bar (int n) {
    symbolicExecution·addNewInput("int_n");
    if (n <= 0) {
        symbolicExecution·updatePathConstraint(true, "n <= 0");
        return 1;
    } else {
        symbolicExecution·updatePathConstraint(false, "n <= 0");
        return 3;
    }
}

```

Then, the code is executed on some random input value, like, for example, $n = 5$. The value 3 is returned by the code and the symbolic execution process produces the following path constraint:

$$PC \equiv \{\neg(n_0 \leq 0)\}$$

The process then flips the constraint corresponding to the last control dependence point met in the concrete execution (the **if** statement) producing the new path constraint

$$PC \equiv \{n_0 \leq 0\}$$

for which $n_0 = 0$ is a satisfying model. The code is thus executed over this new input $n = 0$ and it returns 1. As both branches of the **if** statement have been explored, the process tries to backtrack to the last met control dependence point. As there is none, the process stops as all paths have been analyzed.

Like the static path exploration scheme, the dynamic scheme performs a depth first traversal of the control-flow graph with early pruning of infeasible branches. In presence of loops in the control-flow graph, the process is bounded to a maximal depth [Sen et al., 2005], to guarantee termination.

The interest of using a dynamic instead of static path exploration process is to provide symbolic execution with a direct access to additional runtime information. [Godefroid, 2011] formally compares the respective power of static and dynamic symbolic execution. In realistic code, the symbolic execution process is necessarily imprecise. First, constraints cannot be generated when the tested code calls code whose source is unavailable (like proprietary API, for example). Secondly, programming languages are often complex, evolving and subject to variants, so that the tested code can contain statements for which no constraint translation rules have been developed. Finally, the constraints generated for some statements by the symbolic execution process cannot always be properly handled by the underlying SMT solver, because they are very complex, leading to timeouts, or fall in an undecidable logic. The dynamic path exploration scheme enables one to alleviate such an imprecision, as it has a natural access to concrete values and can use them to replace the parts of the code that cannot be symbolically executed. This process is called *concretization*.

For example, let us consider the following unit of code:

```

int foobar (int n,int m) {
  int p = unhandled(n);
  if (m==p+ n)
    if (n==m)
      return 1;
    else
      return 2;
  else
    return 3;
}

```

where the call to the unhandled function cannot be symbolically executed for some reason. Let us use constraint-based testing with dynamic path exploration over this code. The process starts by executing the code over random inputs, like $n = 3$ and $m = 4$. Let us suppose that `unhandled(3)` returns 8. The else branch of the global **if** statement is taken and the code returns 3. The path constraint is

$$PC \equiv \{\neg(m_0 = \text{unhandled}(n_0) + n_0)\}$$

The process then flips the constraint corresponding to the last control dependence point met in the concrete execution (the global **if** statement) producing the new path constraint

$$PC \equiv \{m_0 = \text{unhandled}(n_0) + n_0\}$$

which cannot be solved because of the unhandled function. Concretization can be used to alleviate this problem by replacing `unhandled( $n_0$ )` by the value 8, saved during the concrete execution where `unhandled(3)` returned 8. The path constraint becomes

$$PC \equiv \{m_0 = 8 + n_0\}$$

for which $n_0 = 3$, $m_0 = 11$ is a satisfying model. The code is thus executed over these new inputs $n = 3$, $m = 11$, the then branch of the global **if** statement and the else branch of the nested **if** statement are taken and the code finally returns 2, producing the following path constraint:

$$PC \equiv \{m_0 = \text{unhandled}(n_0) + n_0, \neg(n_0 = m_0)\}$$

The dynamic path exploration scheme and concretization made it possible to select adequate test cases for two paths. In contrast, the static scheme would have been completely stuck by the presence of the unhandled function. However, concretization is only a fallback that cannot always save the situation, in presence of code that cannot be executed symbolically. To illustrate this limitation, let us proceed with constraint-based testing of our example, and try to generate a test case triggering the path where the code returns 1. The process flips the constraint corresponding to the last control dependence point met in the concrete execution (the nested **if** statement) producing the new path constraint

$$PC \equiv \{m_0 = \text{unhandled}(n_0) + n_0, n_0 = m_0\}$$

which becomes, using concretization:

$$PC \equiv \{m_0 = 8 + n_0, n_0 = m_0\}$$

which is unsatisfiable. However, the path where the code returns 1 may be feasible for $n = m = x$ where x is any integer so that `unhandled( $x$ ) = 0`.

Concretization and its limitations are deeply discussed in [Godefroid, 2011]. The paper shows that a similar (and even more powerful) ability can be conceptually integrated, with some practical difficulties, within the static scheme, using *higher-order test generation*. In this approach, the parts of the code that cannot be directly symbolically executed are replaced by universally quantified uninterpreted functions in the generated SMT instance. Sample concrete input-output values for these parts of code are recorded and integrated in the constraints as well.

It should be noted that both the static and dynamic schemes can be easily adapted to generate the outputs produced by the tested code at the end of the execution of a path, together with the inputs triggering the execution of this path. More generally, the state of any variable during the path execution can be generated. In the static scheme, one needs to represent these outputs (or variable states) by symbolic values which are constrained as well within the path constraint. In the dynamic scheme, one needs to save these output values (or variable states) at the end of each concrete execution. These output or variable values can be useful when the data generated by constraint-based testing is subsequently used for defect detection.

3.2.4 Recent Trends

Since dynamic path exploration and scalable SMT solvers have recently made constraint-based testing a promising approach, academic research over CBT has grown essentially around a set of dedicated tools [Cadar et al., 2011]. DART (Directed Automated Random Testing) [Godefroid et al., 2005] was implemented at Bell Labs. It enables the symbolic execution of C code using a dynamic path exploration process and concretization. It uses `lp_solve` [lp solve, 2014] as underlying solver for linear integer and real arithmetic constraints. CUTE (Concolic Unit Testing Engine) and JCUTE (Cute for Java) [Sen and Agha, 2006] were developed at University of Illinois. These tools extend DART to handle multi-threaded code manipulating pointer-based data structures. CUTE still works on C code while JCUTE enables the testing of Java code. JCUTE was applied notably for finding bugs in the `java.util` package. CREST [Burnim and Sen, 2008] has been developed at the University of California. It uses an engine for testing C code similar to CUTE as well as Yices [Dutertre and de Moura, 2006b] as an underlying constraint solver. Originally built to experiment heuristic strategies for path selection, it is an open-source tool that has been used in various testing scenarios (see e.g. [Kim et al., 2012, Ruse et al., 2010, Xu et al., 2010]). EXE [Cadar et al., 2006] was designed at Stanford University. It enables the symbolic execution of C code. The generated bit-vector and array constraints model the memory at the binary level and are solved using an ad-hoc solver called STP [Ganesh and Dill, 2007]. EXE has been redesigned under the name of KLEE [Cadar et al., 2008], which is open-source since 2009, paving the way for several subsequent applications in several domains (see e.g. [Kuznetsov et al., 2010, Zamfir and Candea, 2010]).

In the same time, constraint-based testing found its way towards an industrial use [Cadar et al., 2011]. Symbolic Path Finder [Păsăreanu et al., 2008] has been under development at Nasa. It enables the symbolic execution of Java bytecode. The generated constraints can mix integer and real constraints. Support for string constraints is reported to be in progress. The underlying solvers used are Choco [choco Team, 2010], IASolver [Hickey, 2000] and CVC3 [Barrett and Tinelli, 2007]. The tool can handle some multi-threaded code manipulating some complex data structures. It has been applied in several internal

Nasa projects (see e.g. [Păsăreanu et al., 2008]) as well as, thanks to its open-source code, in external projects like at MIT [Jayaraman et al., 2009]) or to test web applications at Fujitsu. MIT and IBM researchers used the Apollo [Artzi et al., 2008] dynamic CBT tool to find bugs in web applications. CBT has also been embedded with several testing automation products proposed to developers, such as Parasoft Jtest and Agitar AgitarOne [Xie, 2009]. Microsoft has been developing and using several tools based on CBT, leveraging the Microsoft Z3 SMT solver [De Moura and Bjørner, 2008] to handle the generated constraints. Pex [Tillmann and De Halleux, 2008] enables the symbolic execution of .NET code and uses a dynamic path exploration process. It has been integrated as an add-in to the Microsoft Visual Studio IDE. SAGE (Scalable, Automated, Guided Execution) [Godefroid et al., 2012] offers a constraint-based testing tool that has been scaled and adapted to detect security-related defects in full applications involving hundred of millions of lines of x86 code, like Microsoft Excel. Based on a dynamic path exploration, SAGE has been run continuously for several years on hundreds of applications, representing the most important use of the Z3 solver to date.

The four main challenges [Cadaru and Sen, 2013] tackled by recent research in constraint-based testing are, first, handling the explosion of the number of possible paths, secondly, increasing the scalability limits for solving the generated constraints, thirdly, generating tractable constraints for complex statements and, fourthly, enabling test case selection for concurrent code. These four challenges are discussed in the next paragraphs.

The total number of existing execution paths for an unit of code often grows exponentially with the number of branches in the control-flow graph. The exhaustive depth-first search path exploration scheme of CBT may thus not be able to explore all the paths in reasonable time, even for some small units of code. The early pruning of infeasible branches performed in CBT enables alleviating this problem. Experimental work applying EXE over a set of sample applications [Cadaru et al., 2006] has indeed revealed that often less than 20% of the binary control dependence points met in the control-flow graph had both size feasible. However, even considering the elimination of some paths by early pruning, only a limited subset of the remaining paths can often be explored in the available time budget. As a consequence, several techniques have been developed to drive the exploration towards the most meaningful paths in priority. CREST, KLEE, EXE and PEX have been provided with heuristics guiding the exploration towards still not or less explored regions of the control-flow graph. CUTE [Majumdar and Sen, 2007b], CREST and KLEE have also combined CBT with some kind of random path exploration. In [Boonstoppel et al., 2008], EXE is made able to detect and prune redundant paths. In [Majumdar and Sen, 2007a], CUTE is provided with a mean to abstract from the exploration of inter-procedural paths. In [Collingbourne et al., 2011], KLEE is made able to merge some paths in a single set of generated constraints (but passing in this way the burden to the constraint solver). Several tools, including EXE, SAGE, Symbolic PathFinder [Staats and Păsăreanu, 2010] and CREST [Siddiqui and Khurshid, 2010], have been provided with some kind of parallelization to explore and analyze more than one path concurrently.

Despite the numerous improvements in SMT solver technology in the recent years, discussed in the previous chapter, constraint solving has remained an important limiting factor to CBT scalability. Constraint solving generally constitutes the most time-consuming part of the CBT process. On real application code, the constraints generated by symbolic execution can saturate the underlying solver in some cases. In the one hand, the increasing needs of CBT tools have been and remain one of the driving force for SMT solver research.

In the other hand, techniques have been integrated within CBT tools to alleviate the burden supported by the underlying solver. The typical principle is to exploit the symmetries between the SMT instances generated for each of the different paths, which share many constraints by construction. Irrelevant constraint elimination (notably used in CUTE and EXE) and model caching (notably used in CUTE and KLEE) enable in this way reusing the solutions already obtained for some paths to reduce the number of constraints to solve for the subsequent paths to explore.

The symbolic execution process can generate constraints that are not practically tractable when it has to describe the exact state of the memory after the execution of some complex statements. This problem can be partially alleviated either by generating constraints that only partially describe the effect of these statements over the memory or by using concretization (in the framework of a dynamic path exploration process). However, the less precisely the constraints describe the actual code semantics, the most likely these constraints may not produce a result corresponding in practice to the path for which they were originally generated. The obtained test case could in this way trigger a different path than intended, while feasible paths could be detected as unfeasible and the other way round. Such an imprecision in the test case selection reduces the efficiency of CBT towards a purely random test case selection process. Constraint modeling of pointer operations has notably been a matter of an important work in many tools like CUTE, CREST, EXE, KLEE and SAGE. Pex is able to generate constraints for almost all .NET statements. It uses approximations for Σ -theories not handled by Z3 such as strings.

Several tools like Symbolic Path Finder or CUTE enable test case selection for concurrent code, which is frequent in real applications. Symbolic Path Finder combines symbolic execution with model-checking and takes advantage of the model-checker's built-in abilities to handle multi-threaded code. In CUTE, symbolic execution is combined with so-called dynamic partial order reduction in order to generate test inputs and thread schedules at the same time.

Finally, it should be noted that the constraint-based testing process can search for inputs triggering those paths ending with a thrown runtime error. Given a path leading to a statement able to produce a runtime error, the constraints are generated so that the memory state before the statement execution will mandatorily trigger the error. CBT can thus provide not only test cases covering most of the different normal behaviors of the code, to be verified with respect to the code specification, but also provide inputs triggering anormal code behaviors, which are direct indications of code defects. Such a bug detection approach has particularly been leveraged in EXE.

If constraint-based testing has still had only a limited impact over software engineering practice [Cadar et al., 2011], current techniques have proven [Cadar and Sen, 2013] their ability to find effectively defects in various kinds of software in an industrial context. Further improvements in path selection and constraint generation would notably be beneficent to scale the approach on very large units of code, as well as on a more various range of programming statements and datatypes. Similarly, better and wider solving capabilities would be useful for the same reasons, notably considering non-linear mathematical constraints, string constraints and combination of both, which are important for testing of real code [Cadar et al., 2011]. Despite the general undecidability of such constraints, work is on-the-go for providing efficient solving capabilities (see e.g. [Jovanovic and de Moura, 2012] for non-linear arithmetics and e.g. [Zheng et al., 2013, Bjørner et al., 2009, Redelinghuys et al., 2012, Saxena et al., 2010, Tateishi et al., 2013, Shannon et al.,

2009, Liang et al., 2014] for strings).

3.2.5 Alternative Automatic Test Input Generation Techniques

The most simple alternate approach to constraint-based testing is *random testing* [Duran and Ntafos, 1984]. This process basically selects test inputs randomly, has very low computational needs and is reputed to work well for simple units of code [McMinn, 2004]. With random testing, the probability of particular path to be tested equals the proportion of test inputs triggering it among all the possible test inputs. As soon as some zones in the control-flow graph are touched by paths triggered by a reduced proportion of test inputs, random testing may miss these zones in the allowed time budget. Techniques directing the selection towards a proper coverage of the control-flow graph, such as constraint-based testing, enable solving this problem.

Search-based testing (SBT) [McMinn, 2004, McMinn, 2011] is another directed approach for the automation of test input generation. In a nutshell, SBT uses common optimization heuristics (such as evolutionary algorithms) to search the set of possible test inputs for test cases covering execution paths in the tested code. Similarly to constraint-based testing, the search-based approach has received a renewed interest in recent years [McMinn, 2011]. At our best knowledge, there is no proper study comparing SBT with CBT. However, several very recent works, including [Malburg and Fraser, 2011, Baars et al., 2011, Sakti et al., 2012, Pablo et al., 2014], have combined CBT and SBT to offer a better testing performance.

Both constraint-based and search-based testing are *white-box* and *structural* approaches for test input generation [Zhu et al., 1997].

Black-box testing approaches use an analysis of the code specification to generate test inputs, while white-box ones, like CBT, use an analysis of the code itself to do so. Combining both approaches and taking advantage of information both from the specification and the code, in so-called grey-box testing, is often recommended to make testing even more effective. The availability of a minimal formalization for the specification of the tested code is a necessary condition to enable the automation of black-box testing.

Structural white-box techniques, like CBT, try to generate test inputs that cover a representative part of the code *structure*, i.e. that cover a representative set of paths in the code. Probably the most typical example of a *non-structural* white-box testing approach is *mutation testing*. In this approach, the quality of a test set is measured as its ability to detect the difference of behavior between the tested code and some error-seeded versions of it, called mutants. Some automated techniques (see e.g. [Fraser and Zeller, 2010, Jia and Harman, 2011]) have adapted CBT and SBT to generate test inputs that are good mutant detectors.

Chapter 4

Constraint-Based Testing of SQL Applications: a State of the Art

Contents

4.1	SQL Normalization into Native Code	99
4.2	Query Results as Input Arrays	100
4.3	Related Research over SQL Application Testing	102
4.3.1	Requirement and Opportunity Analysis	102
4.3.2	Selecting Inputs for Apps Working on an Existing Database . .	102
4.3.3	Code Coverage Criteria for SQL Applications	102
4.3.4	Code-Independent Test Database Generation	103
4.3.5	Other Testing Approaches and Objectives	105

IN this thesis, we generalize the constraint-based testing approach to select test cases for SQL applications. SQL applications are units of code mixing native code with SQL statements. In this chapter, we review the related work concerning SQL application testing.

The principle of using a constraint-based approach for testing SQL applications is recent. Two main research approaches have been proposed to do so: the first one is based on the normalization of SQL into native code and is presented in section 4.1, the second one considers results of SQL queries as input arrays for the code and is presented in section 4.2. The last section of the chapter (section 4.3) presents other related research around testing of SQL applications.

4.1 SQL Normalization into Native Code

An early work that has considered test input generation in presence of SQL statements in the tested code is [Chan and Cheung, 1999]. The paper proposes to translate every SQL statement into the native language of the tested code. The article considers C++ as a target language. It defines a C++ class for representing relations and details a mapping between the operators of the relational algebra and some C++ code manipulating relation-typed objects. A translation to C++ is also provided for some other SQL mechanisms, like row sorting. The paper advocates that existing white-box testing approaches could then be applied seamlessly to the “normalized” code. The authors also discuss that replacing a single SQL statement by a piece of code that simulates (at least a part of) its execution increases the number of paths to be explored by a white-box testing approach, compared to the original code. As an example, the following code (written using the syntax proposed in [Chan and Cheung, 1999]), containing two execution paths:

```
result = SELECT *FROM client WHERE id = 0 ;
if (result →isEmpty()) {
    return 1;
} else {
    return 2;
}
```

becomes, after translating the SQL code as proposed in [Chan and Cheung, 1999]:

```
Relation* client = input(); // Input content of the client table
Relation* result = new Relation;
for (int i=0; i< client·getNoOfTuples(); i++ ) { // Iterating over the client table
    if (client·getTuple(i)·getAttrValue(ID) == 0) { // Selecting tuples
        result →insert(client·getTuple(i)); // Insert selected tuple in result set
    }
}
if (result →isEmpty()) {
    return 1;
} else {
    return 2;
}
```

which defines an infinite number of execution paths, as the added code contains a loop to enumerate the unbounded content of the database table.

A very recent and conceptually similar technique is proposed in [Pan et al., 2013b], which deepens the work of [Taneja et al., 2010]. Compared to [Chan and Cheung, 1999], this approach essentially proposes a true automation of the process, for C# code. Given the relational schema of the database manipulated by the tested application, the technique proposes a mapping making it possible to derive automatically a set of C# classes implementing this schema. The content of the database can then be encoded by a set of objects implementing these classes. These objects are encapsulated within a DatabaseState object, which constitutes a functional C# implementation of the database. It is able to receive and apply data read and write operations, as in the real database. The C# classes generated by the mapping are indeed automatically provided with accessor and modifiers methods, together with code able to verify that the content of this mock database does not violate the integrity constraints defined in the original schema.

The code under test is then modified to interact with the mock database instead of with the real one. First, the tested code is changed to receive a `DatabaseState` object as input instead of a connection to the real database. Secondly, the calls to the CLI API of the DBMS present in the tested code are replaced by calls to mock methods, receiving the `DatabaseState` object and the symbolic SQL string built by the code as call parameters. These mock methods parse the symbolic SQL code, which is translated into equivalent C# code that performs equivalent operations over the `DatabaseState` object.

The code modified to interact with the mock database is then provided to the Pex constraint-based testing tool from Microsoft [Tillmann and De Halleux, 2008], which performs symbolic execution using a dynamic path exploration process. This tool generates in this way test inputs, including an initial content for the `DatabaseState` object, and thus for the original database as well.

The approach is tested by searching test inputs for 39 Java methods (translated to C# using an automated compiler) for a total of 1279 LOCs, mixing Java statements with one or a few simple or compound `SELECT` SQL statements. These methods were extracted from the `UnixUsage`¹ and `RiskIt`² open-source applications, available on the web. `UnixUsage` is a monitoring application for Unix, manipulating a database with eight tables and thirty one attributes. `RiskIt` is an insurance rate estimation application, manipulating a database with thirteen tables and fifty-seven attributes. The results show the ability to generate test inputs offering a strong coverage of the code statements, with a mean analysis time of 2m05s per method (shortest: 58s, longest: 4m21s). The code of some Java catch statements could not be covered because the technique is unable to generate inputs that trigger the throwing of runtime exceptions by the database connection.

4.2 Query Results as Input Arrays

The JCUTE constraint-based testing tool [Sen and Agha, 2006] enables automatic test case selection for Java code, using a dynamic path exploration process and concretization. In presence of SQL code in the tested application, JCUTE can at best ignore it and use concretization for each datum read from the database appearing in the path constraint.

In [Emmi et al., 2007], the authors integrate a technique in JCUTE that alleviates this problem in presence of simple `SELECT` statements in the tested code. This technique is detailed for a small imperative language with assignment, conditional, go to and SQL `SELECT` statements. The principle is to consider the result of any `SELECT` statement as a new code input typed as an array. The size of each of these input arrays, as well as the content of each of their cells can then be accessed in the normal code. The content of the cells can be either integers or strings. As a consequence, the path constraint generated by classical JCUTE will thus contain integer and string constraints over the size and content of the input arrays. Additional constraints are then added to this path constraint, enforcing that each row in an input array makes true the condition of the `WHERE` clause from its corresponding `SELECT`. The resulting path constraint is solved for finding a test case, including values for the input arrays, by splitting it into a linear integer part and a string part (if possible). The integer part is solved using the `lpsolve` [lp solve, 2014] solver. An ad-hoc string solver is used for the string part. It handles string (dis)equality and string membership in a regular language. The constraints are also preprocessed to

¹<http://sourceforge.net/projects/se549unixusage>

²<https://riskitinsurance.svn.sourceforge.net>

account for the presence of *null* values, which can make some predicates have an unknown value. The approach handles dynamic SQL code in a simple way: the concrete value of the dynamically built SQL string sent to the DBMS, obtained through concretization, is parsed, in order to obtain the WHERE clause necessary for constraint generation. The approach is evaluated by searching inputs for a 16-LOCs Java method involving the following SELECT query:

```
SELECT *  
FROM cur  
WHERE cur_namespace=0 LIMIT 1 OFFSET $randomCurIndex
```

where “**LIMIT 1 OFFSET \$randomCurIndex**” force the query to return a single random row among those enforcing the WHERE clause.

We now briefly demonstrate the technique from [Emmi et al., 2007] over the following pseudo-code (written using the syntax proposed by [Emmi et al., 2007]):

```
1: $client := query(SELECT *FROM client WHERE name = 'Michael');  
2: if ($client · size ()=2) go to 4  
3: return 2;  
4: return 1;
```

Let us suppose that the dynamic path exploration process starts by running the code on an empty database. The code returns 1 and the path constraint is:

$$PC \equiv \{\neg(\text{size}(\text{client}_0) = 2)\}$$

where client_0 is an input array. In order to obtain an input database triggering the path where the code returns 2, the technique flips the last constraint and adds the constraints from the WHERE clause, obtaining the following path constraint:

$$\text{size}(\text{client}_0) = 2 \quad \wedge \quad \forall i \in \{1, \dots, \text{size}(\text{client}_0)\}, \text{client}_0[i, \text{name}] = \text{Michael}$$

By solving this constraint, the technique is now able to insert two rows into the client table, with value Michael for the name attribute. The unconstrained attributes are filled with random values. The code is then run over this new database content and the dynamic path exploration process can proceed along the path where the code returns 2.

Authors of [Pan et al., 2013b] compare their normalization approach with (a simulation of) the input arrays approach of [Emmi et al., 2007]. The normalization approach is measured to offer a 25.65% increase in statement coverage over the 39 methods tested in [Pan et al., 2013b]. This increase is said to have two main reasons. First, the input arrays approach does not take the database schema integrity constraints into account. This can lead to the generation of an invalid database input content, by the way reducing the code coverage. Secondly, using concrete string values for the WHERE clauses to derive constraints means applying concretization for the variables used to build this query. By nature, concretization may produce constraints that do not drive properly the execution along the intended path, randomizing the process and thus possibly reducing the code coverage.

4.3 Related Research over SQL Application Testing

4.3.1 Requirement and Opportunity Analysis

[Haller, 2009] provides a requirement analysis for the process of testing SQL applications, using a white-box test case selection approach. It considers every step of the typical white-box testing process, as usually defined for classical applications, and discusses the interest and challenges to adapt the classical testing concepts and automation techniques to SQL apps. The paper is based on the author’s experience in practical SQL application development. It aims at providing a clearer path to the recent field of SQL apps testing, driving it towards a practical usefulness in commercial software engineering.

In [Haller, 2010], the same author provides decision-making elements to industry testers for choosing a method to provide a database with test data. The opportunity to use an automatic test data generator is compared with other approaches (existing database snapshot, manual insertion, etc.) in several scenarios (privacy of existing data, need for mass testing, etc.).

4.3.2 Selecting Inputs for Apps Working on an Existing Database

SQL applications can be required to work with data already stored in an existing database. Some papers study the particular situation where traditional test inputs must be selected for code manipulating in parallel an existing database with a known input content.

[Li and Csallner, 2010] aims at testing units of code manipulating a relational database whose initial content is well-known and fixed. The tested code receives one SQL SELECT statement to execute over this database (or at least its WHERE clause) as an input string. The paper proposes a constraint-based testing approach able to generate a set of input SQL queries, optimizing the coverage of the tested code. A prototype tool implementing the proposed technique is said to be under development.

[Pan et al., 2011b] generalizes the Microsoft Pex constraint-based testing tool [Tillmann and De Halleux, 2008] to generate traditional program inputs (function parameters, etc.) for units of code performing SELECT statements over a database whose initial content is known. The approach enables Pex to analyze the WHERE clause of the executed SELECT statements and to query the database content, in order to help the generation of program inputs triggering some expected paths. The technique is evaluated over 45 Java methods from the UnixUsage and RiskIt open-source applications, already described in the previous paragraph for [Pan et al., 2013b]. The code coverage is measured to increase by 31.1% compared to the case where the Pex tool is run alone, using concretization. The proposed work is extended in [Pan et al., 2014].

4.3.3 Code Coverage Criteria for SQL Applications

Path explosion is an important challenge faced by constraint-based testing approaches. As only a limited number of paths can often be explored in the available time budget, code coverage criteria provide a rationale to prioritize the paths to explore. A code coverage criterion defines which part of the code should be covered by a test set to consider this test set as adequate. While many code coverage criteria have been defined in the history of testing (see e.g. [Zhu et al., 1997] for a review), several works have considered advanced

criteria for SQL applications.

[Kapfhammer and Soffa, 2003] proposes a family of test adequacy criteria based on the dataflow associated with the different entities present in a relational database. The criteria have been designed to capture the interaction between the code and the database at different levels of granularity. They aim at guaranteeing test sets able to detect violations of data integrity.

[Suárez-Cabal and Tuya, 2004] describes a coverage criterion for SQL SELECT queries, aiming at covering different truth values for the boolean conditions involved in the query. The authors push their research forward in [Suárez-Cabal and Tuya, 2009], notably considering more complex SQL queries. A more scalable criterion is proposed by the same authors in [Tuya et al., 2010].

[Willmor and Embury, 2005] introduces another family of test adequacy criteria for database system, notably taking into account transactions and involving a reasoning over dynamic database states in the application.

Another criterion, accounting for the dynamic building of possibly different SQL statements by the application, using string operations, is presented in [Halfond and Orso, 2006].

[Pan et al., 2011a] presents a constraint-based testing approach to generate database states for SQL applications which are sufficient to satisfy some particular code coverage criteria. These criteria aim at testing boundary points and the instantiation of the different clauses in the appearing logical expressions.

[Zhou and Frankl, 2012] describes an empirical analysis performed on SQL applications. This analysis studies the relation between coverage and effectiveness and compares different coverage criteria, in the framework of Java/JDBC programs.

4.3.4 Code-Independent Test Database Generation

For a long time (see e.g. [Lyons, 1977] for an early approach), a substantial amount of work has been done on how to generate test database content exhibiting some desirable properties, given only the database schema and possibly some queries to be executed over the database. These approaches work thus without considering the control flow of the programs manipulating the database. We focus here on the most recent approaches aimed generating data for testing SQL applications.

In [Robbert and Maryanski, 1991], the authors describe a C++ program able to generate automatically test plans for database applications. The program analyzes the database schema and produce automatically a textual test plan, issuing a sequence of instructions (but no concrete test case) for manual testing of the system.

[Binnig et al., 2007] introduces reverse query processing (RQP). This process, based on an inverted relational algebra, receives a SQL SELECT query and a result for this query and and it produces a database content that could have triggered that result for that query. A tool implementing the approach was evaluated over 20 business-oriented complex queries on a database with 8 tables. The reverse query processing approach was notably leveraged in [Binnig et al., 2006] in the context of testing database applications. It was generalized in [Binnig et al., 2008] to generate a single database content for several queries and their expected result together.

[Zhang et al., 2001] studies the constraint-based selection of database instances enforcing the semantics of some SQL statements. The approach generates constraints that represent

a property against which the program is tested and whose solutions are database instances. Such instances can be obtained by leveraging existing constraint solvers.

The authors of [Chays et al., 2004] introduce a multi-usage tool for testing SQL applications called AGENDA. The tool is able to generate an input database content enforcing the schema constraints. It employs either user-provided data or boundary values obtained heuristically by analyzing the database schema and the SQL statements to be executed on this database. The tool use similar heuristics to generate test values for the parametric part of the WHERE and SET clause of the SQL statements present in an application manipulating the database. Finally it enables logging and analyzing the changes operated by an application over a database. To original version of the tool targets simple applications involving single queries. [Deng et al., 2005] make it possible to handle SQL transactions with multiple queries. The tool was also improved in [Chays et al., 2008], in order to avoid the generation of a database content making the SQL statements from the tested code return empty results or violate schema constraints.

[Willmor and Embury, 2006] proposes a language, inspired by SQL, enabling database application testers to describe the database content required for their test in a declarative way. An algorithm is described to modify the database content so that specified content required is obtained. A tool implementing the approach is evaluated for 20 tests cases over a database with 6 tables, 22 attributes and 7 primary key constraints and 6 foreign key constraints.

In [Veanes et al., 2009] and [Veanes et al., 2010], the authors present the Microsoft Qex tool (Query EXplorer). Given the database schema, a SELECT query and a condition over the result of this query (like e.g. the result must be empty), the tool aims at producing an input database so that executing the query over this database will produce a result satisfying the condition. In order to do so, the input content of each table in the database is modeled as a fixed-size list of tuples satisfying the schema integrity constraints. The result of the query is modeled as the combination of several operators acting on these input lists. An SMT instance stating that this combination satisfies the condition over the query result is then passed to the Z3 SMT solver. A satisfying model for this instance constitutes the searched input database. The solver is first given an instance where the size of the input lists is 1. If no model is found, the solver is provided with a new instance where the size has been increased. The process is repeated until the available time budget is consumed. In order to solve the constraints, the solver is provided with a set of axioms for list operators. Strings are modeled using bit-vectors. The tool is evaluated over a collection of sample queries.

[de la Riva et al., 2010] considers the generation of a test database making it possible to test SQL SELECT queries according to the coverage criterion proposed by the authors in [Tuya et al., 2010]. The database schema and the test requirements obtained by applying the criterion to the queries are modeled in the Alloy language. Solving the Alloy constraints enables returning a consistent test database satisfying the test requirements.

Some other works aim at generating data for testing the DBMS or the database schema. Some of these works have taken advantage of a constraint-based approach, where parts of the SQL semantics are transformed into propositional logic [Binnig et al., 2007] or Alloy constraints [Khalek et al., 2008] (the Alloy model represents here several CREATE and a single SELECT statement).

4.3.5 Other Testing Approaches and Objectives

Several works have considered mutation testing of SQL applications.

The SQLMutation tool introduced in [Tuya et al., 2006] enables one to create mutants of SQL SELECT queries.

In [Zhou and Frankl, 2009], the authors develop a research infrastructure to compare the mutant-killing effectiveness of different test generation techniques for SQL applications.

The authors of [Pan et al., 2013a] adapt their constraint-based tool from [Pan et al., 2013b] to mutation testing. This tool, presented at the beginning of this chapter, performs SQL code normalization to enable SQL application testing. Considering the normalized code, the approach leverages off-the-shelf mutant generators to produce mutants. The symbolic execution engine of Microsoft Pex is then employed over the mutants, aiming at the generation of mutant-killing test cases. The technique is evaluated over 15 Java methods from the UnixUsage and RiskIt open-source applications, improving the mutant killing ability by 16.6% compared to some existing pure constraint-based approaches [Emmi et al., 2007, Taneja et al., 2010].

Finally, the authors of [Appelt et al., 2014] have investigated security testing of SQL applications, where they intend to generate inputs triggering dangerous SQL injection attacks.

Part II

Relational Testing of SQL Applications

*“To Harry James Potter, I leave the Snitch he caught in his first Quidditch match at Hogwarts,
as a reminder of the rewards of perseverance and skill.”*

ALBUS DUMBLEDORE in *Harry Potter and the Deathly Hallows* by J.K. Rowling

Chapter 5

Relational Testing: an Introduction

Contents

5.1	Motivations	111
5.2	Principle	113
5.3	Methodology	115

IN the forthcoming second part of this manuscript, we detail the original research work constituting the material of the thesis. This description is logically based on the contextualization proposed in the four previous chapters composing the first part of the manuscript.

This first chapter of the second part offers an introduction to relational testing, the research approach developed in this work. It motivates the research (section 5.1), describes the principle of relational testing (section 5.2) and presents the followed methodological process (section 5.3). This last process then serves as a basis to structure the description of relational testing made in the following chapters.

5.1 Motivations

In this research, we propose a new technique, called relational testing, to extend the constraint-based testing state-of-the-art with the ability to handle SQL applications. In this subsection, we motivate our research by considering its potential impacts, expected challenges and conceptual differences with related work.

In current software engineering practice, testing remains the primary approach to detect defects in software. But testing also remains a complex, repetitive and thus costly task. As a consequence, test automation has been widely investigated as an interesting mean to improve software reliability and reduce development cost. Among the many aspects of testing that can be automated, barely few works have studied how to generate automatically inputs to test the correct interaction between a unit of code and a relational database. As databases are ubiquitous in software, there is thus room for research improving both software reliability and cost in a new and meaningful way. More fundamentally, as more and more data are collected and processed about all aspects of personal, social and economical life, such research, providing an effective mean to locate sources of data corruptions, would avoid increasingly damaging consequences for the society.

Constraint-based testing for imperative code has been widely studied, as well as generalized to lower-level or higher-level programming paradigms, such as for x86 assembly or Java and C# object-oriented code. In SQL applications, such classical code is mixed with SQL DML code to interact with a relational database, whose schema has been previously defined using SQL DDL code. Enabling test input generation for SQL applications is a non-trivial extension of known constraint-based generation techniques, both from a theoretical and practical standpoint.

From a theoretical standpoint, generating inputs triggering a particular execution path, in presence of purely relational SQL statements in the path, is not a computable problem in general [Veanes et al., 2010]. Given any query restricted to the relational-complete part of SQL, the problem of finding a database content so that executing the query on this content will return a non-empty result has been proven recursively enumerable, but not recursive [Abiteboul et al., 1995]. This semi-decidable problem is known as the *query satisfiability problem*. As generating inputs for any path containing SQL code notably means being able to find a database input making the arbitrary queries in the code return non-empty results, input generation for paths containing SQL is not decidable either. Considering a subset of the SQL language richer than its relational-complete part will make the problem even more difficult. For example, if one allows SQL code performing non-linear operations over integer values, finding path inputs will then require to write and then solve constraint written in Peano arithmetic, which is an undecidable problem in general [Gödel, 1931].

In practice, computer scientists have never been afraid of developing good approximate algorithms for problems that are not decidable. Nevertheless, even from a practical standpoint, generalizing constraint-based testing to SQL applications is a challenging task. A central reason for this is that major issues remain to be solved to enable the symbolic execution of SQL code, which is the core feature necessary for a proper constraint-based testing of SQL applications. Solving these issues, detailed in the next two paragraphs, constitutes the central motivation for the research presented here.

In an SQL application, the database can be seen as a particular kind of container for

some of the values manipulated by the program. During symbolic execution, these values must thus be represented symbolically and subsequently constrained to enable the proper generation of test data, including an input database content. The symbolic representation of these values raises difficulties as the database is a container of particularly complex shape: its content must obey the DDL schema of the database. This schema organises the data in tables containing relations with no limit on the potential number of tuples. The schema also defines a set of integrity constraints that must be satisfied by the relations in the tables. These constraints are particularly complex as they are first-order logic constraints.

SQL applications interact with the database by using SQL SELECT queries, to copy values from the database into the application's variables, as well as using INSERT, UPDATE and DELETE statements, to modify the content of the database tables, typically in function of the value of the application's variables. Symbolic execution of such DML statements raises difficulties due to their complex behavior. Firstly, SQL is a declarative language, where symbolic execution has been designed in the context of imperative code. SQL statements express the desired action over the content of the database tables, but they do not explicit the (often complex) control-flow followed in practice by the DBMS to compute this action. Secondly, the execution of DML INSERT, UPDATE or DELETE statements by the DBMS does not only consist in modifying the content of the database, but also in checking that these modifications do not let the database in a state where the integrity constraints defined in the schema are violated. If an integrity constraint violation is detected, the DML statement execution fails and the database remains unmodified. As a consequence, during symbolic execution, each INSERT, UPDATE or DELETE statement will have to be treated as an if-then-else statement with a particularly complex condition:

```

if (Applications variables and Database are in a state
      where the SQL statement will not violate any constraint) {
  Execute the SQL statement!
} else {
  Signal a constraint violation !
}

```

Other approaches to constraint-based testing of SQL applications either normalize the SQL code into native code, or they model query results as input arrays. In the first-case, normalization provides an actual way to represent symbolically the values in the database and to constrain them properly, by symbolically executing the normalized SQL code. However, the need to replace each single SQL statement by complex native code, involving conditionals and loops, to simulate a mock database, may increase dramatically the number of paths to explore in the code to find input data. This phenomenon must not be neglected as it may degrade the testing performance. Firstly, because the number of paths to explore typically grows exponentially with the number of static branches in the code. Secondly, because path explosion is one of the main challenges still faced by modern constraint-based testing. In the second case, modeling query result as input arrays enables representing symbolically the database content in an indirect way (as the constrained symbols only represent the extracted data). However, it does not provide any mean to test applications that do not only read into the database, but also write data in it using INSERT, UPDATE and DELETE statements.

The relational testing approach introduced in this work defines a direct symbolic execution

for SQL, able to represent symbolically and to constrain properly all the values stored in the database. This approach is notably original as it enables test generation even in presence of INSERT, UPDATE and DELETE statements and without increasing the number of paths to explore. We detail the principle of relational testing of SQL applications in the next subsection.

5.2 Principle

Following the concept of relational variables introduced in our description of the relational model in Chapter 1, the principle of relational testing is to consider every table in the database as an application's variable typed as a relation. An initial relation value is received as input by the application for each of these relational variables. Each SQL statement in the application can then be considered to be a relational operation over these relational variables, as well as over the classical application variables.

By applying the classical symbolic execution process to this relational version of the application, relational testing can derive the necessary path constraint over the values of both the application's relational and classical inputs. The generated path constraint will thus include symbols representing classical application values as well as symbols representing the relations in the database tables. Furthermore, relational testing combines the generated path constraint with the necessary schema constraints over the defined relational symbols, in order to enforce database integrity all along the path execution. The result is a complex constraint system that mixes classical constraints over the application's classical inputs with relational constraints over the relational inputs. These constraints can be expressed either in the Alloy language, or into the SMT-Lib language. Each solution to the obtained combined constraint system describes test data, *including* a valid initial content for each table in the database, such that when the application is executed with respect to these input values, the execution will follow the path represented by the constraint system.

We now illustrate conceptually the relational symbolic execution mechanism that we introduce with relational testing, over a sample SQL application. The schema of the database manipulated by this application is defined by the following DDL code:

```

1 CREATE TABLE account (
2 number INTEGER NOT NULL,
3 balance INTEGER NOT NULL,
4 CONSTRAINT accPK PRIMARY KEY (number),
5 );

```

The database contains one table representing bank accounts. Each account has a unique number and a balance value. The SQL application itself is the following Java method, retrieving the balance value associated to an input account number. If no account exists with this number, a new one is created and a balance value of 0 is returned:

```

1 int getBalance (Connection con,int accountNumber) throws SQLException {
2   ResultSet result = con.createStatement().executeQuery(
3     "SELECT balance
4     FROM account
5     WHERE number="+ accountNumber);
6   if (result.next()) {

```

```

7  | return result.getInt("balance");
8  | } else {
9  |   con.createStatement().execute(
10 |     "INSERT INTO account VALUES (" + accountNumber + ",0)");
11 |   return 0;
12 | } }

```

Let us demonstrate the relational symbolic execution process of relational testing, over the execution path where the else branch of the if statement is taken and where the INSERT statement does not throw any exception.

At the beginning of the relational symbolic execution process, the memory is initialized as:

$$M \equiv \{account \mapsto r_0, accountNumber \mapsto n_0\}$$

where account is the relational variable representing the account table and $r_0 \subset \mathbb{Z} \times \mathbb{Z}$ the symbol representing the input relation received for this variable. The path constraint is initialized with a constraint enforcing that r_0 satisfies the database schema, i.e. the primary key constraint:

$$PC \equiv \{\forall i_{num}, \forall i_{bal}, \forall j_{bal}, ((r_0(i_{num}, i_{bal}) \wedge r_0(i_{num}, j_{bal})) \implies i_{bal} = j_{bal})\}$$

The constraint is quantified over all the tuples in the input relation. Such a quantification will always appear in the path constraints generated by relational testing, as the number of tuples necessary to trigger a considered path is theoretically unbounded and unknown a priori. After executing symbolically the SELECT statement, the memory becomes:

$$M \equiv \{account \mapsto r_0, accountNumber \mapsto n_0, result = \pi_{\mathbf{balance}}(\sigma_{P_{number=n_0}}(r_0))\}$$

where result is a relational variable representing the relation extracted by the SELECT query. It is computed from r_0 using the adequate relational operators describing the executed query, as discussed in chapter 1. In order to take the else branch of the if statement, the relation stored in result should have an empty cardinality. The path constraint is thus updated as:

$$PC \equiv \left\{ \begin{array}{l} \forall i_{num}, \forall i_{bal}, \forall j_{bal}, ((r_0(i_{num}, i_{bal}) \wedge r_0(i_{num}, j_{bal})) \implies i_{bal} = j_{bal}) \quad , \quad \\ \#(\pi_{\mathbf{balance}}(\sigma_{P_{number=n_0}}(r_0))) = 0 \end{array} \right\}$$

Then, as the INSERT statement should not throw any exception, the path constraint is updated with a constraint enforcing that the memory state is so that no schema constraint will be violated by this INSERT. In this case, the added constraint must enforce that the account table does not already contain a row with a number equal to the one of the inserted row:

$$PC \equiv \left\{ \begin{array}{l} \forall i_{num}, \forall i_{bal}, \forall j_{bal}, ((r_0(i_{num}, i_{bal}) \wedge r_0(i_{num}, j_{bal})) \implies i_{bal} = j_{bal}) \quad , \quad \\ \#(\pi_{\mathbf{balance}}(\sigma_{P_{number=n_0}}(r_0))) = 0 \quad , \quad \\ \forall i_{num}, \forall i_{bal}, (r_0(i_{num}, i_{bal}) \implies \neg(i_{num} = n_0)) \end{array} \right\}$$

The symbolic execution of the INSERT statement updates the memory, which becomes:

$$M \equiv \{account \mapsto r_0 \cup \{(n_0, 0)\}, accountNumber \mapsto n_0, result = \pi_{\mathbf{balance}}(\sigma_{P_{number=n_0}}(r_0))\}$$

As a computation termination point is met, the path constraint can be solved to find an input value for r_0 and n_0 . The obtained solution will provide a valid input content for

the account table and an input value for the `accountNumber` parameter, triggering the considered path.

The constraints generated here by relational symbolic execution are expressed in quantified first-order logic modulo a Σ -theory for relations and integers. In presence of additional variables that would not be typed as relations or integers, like strings or bit-vectors variables, constraints expressed modulo the corresponding Σ -theories would have appeared in the generated constraints.

First-order integer and relational constraints can be expressed naturally in the Alloy logic and solved using the Alloy solver. They can also be translated into the SMT-Lib AUFLIA logic, using the equivalence between the relational algebra and the relational calculus to translate the relational part into first-order constraints. As studied in [El Ghazi and Taghdiri, 2011], translating quantified relational and integer constraints into SMT-Lib and solving them using the Z3 solver enables detecting the unsatisfiability of the constraints (and thus the unfeasibility of paths and branches), which is impossible using the Alloy solver. However, [El Ghazi and Taghdiri, 2011] reports that Z3 was rather limited for finding models satisfying the constraints (and thus generating test inputs), compared to the Alloy solver, due to the presence of quantifiers. However, since the publication of [El Ghazi and Taghdiri, 2011], new model-based quantifier instantiation approaches have been integrated into Z3, promising better model finding capabilities for quantified constraints. Which solver is possibly the best fitted for relational testing and with which practical limits are thus open research questions, to be answered through experimentation. It should however be noted that, in contraposition to modern SMT solvers, the Alloy solver cannot handle other Σ -theories than relations and integers, which would limit the potential scope of applications handled by relational testing. Z3, on the other hand, handles many different Σ -theories [De Moura and Bjørner, 2008].

As a conclusion, by returning to the logical roots of SQL, i.e. relational algebra and calculus, relational testing enables the direct symbolic execution of the core relational part of SQL into constraints, which can be incorporated seamlessly within classical path constraints. This makes constraint-based testing able to generate test data both in presence of read and write SQL operations, which enables a wider applicability for the technique. This also makes constraint-based testing able to generate test data without normalizing the SQL code and increasing the number of paths to explore. This could lead to improved performance for the technique, which should be verified experimentally as an open research question. As the relational path constraints to be solved for generating test data are expressed in a quantified (and thus not decidable) logic, relational testing is coherent with the test generation problem for SQL applications being not decidable.

5.3 Methodology

After having established the main motivations and principle of our research, we take advantage of this last introductory subsection to present our methodological approach. First, we have carefully selected the framework of our study. In order to do so, we have defined SimpleDB, a subset of Java/SQL to write SQL applications. SimpleDB enables writing simple Java methods with JDBC/SQL. It can be used to model database interaction use cases met in transactional software. Then, we have built a symbolic execution algorithm for SimpleDB code, based on the principle of relational testing. A first

version of this algorithm produces Alloy constraints and a second one SMT-Lib constraints. The two versions have been used to generate test inputs for a set of hand-crafted SimpleDB applications, enabling a proper experimental evaluation of the Alloy and Z3 solvers for relational testing. As the Z3-powered SMT-Lib version revealed to be the only practical one, further scalability measures were taken to estimate the current practical limits of this version, and it was then evaluated over open-source code available on the web. This last evaluation enabled a performance comparison with related work based on normalization of SQL code.

These successive methodological steps serve as a basis to structure the description of our work made in the following chapters. Chapter 6 presents the SimpleDB language. Chapter 7 details the Alloy and SMT-Lib symbolic execution algorithms. Chapter 8 presents the experimental work performed using these algorithms. Chapter 9 proposes a discussion over the relational testing approach, based on the obtained experimental results and a proper comparison with related work.

The work presented in these chapters has been published in [Marcozzi et al., 2012, Marcozzi et al., 2013]. It has been invited for submission to the SCP journal. As the review process is in progress, a preliminary version is available to the reader as a technical report [Marcozzi et al., 2014b].

Chapter 6

SimpleDB, a Core Language for Studying SQL Applications

Contents

6.1	Defining a Core Java/SQL for OLTP Use Cases	119
6.2	A Formal Definition for SimpleDB	120
6.2.1	SimpleDB Program	120
6.2.2	Database Schema	120
6.2.3	Method Signature and Body	122

IN this chapter, we present and detail the SimpleDB language, which constitute our framework for studying the relational testing approach. SimpleDB is a formal subset of Java and SQL enabling one to write SQL applications in the context of transactional software.

Section 6.1 proposes a general description of SimpleDB and justifies the choices made to build this study framework. Section 6.2 offers a formal and precise description of the SimpleDB language, using a BNF grammar.

6.1 Defining a Core Java/SQL for OLTP Use Cases

SQL applications can be written in many different contexts and programming languages, using different DBMS interfaces, and taking advantage of the numerous features of the various SQL dialects. This strong variability makes necessary to define a more restricted framework to study relational testing. Methodologically, this framework should provide a well-balanced compromise between a large scope for the studied SQL applications and the time constraints imposed on the study.

Practically, we have chosen to define a formal subset of the language made of Java empowered by JDBC and ISO SQL, which are all very popular technologies. We have then built a complete and coherent relational testing algorithm, able to handle any SQL application written in this sublanguage. The defined sublanguage was called SimpleDB. SimpleDB contains base SQL primitives as well as a Turing-complete subset of Java. Constraint-based testing of more complex Java structures than those included in SimpleDB is orthogonal to the work of this thesis and has been studied elsewhere.

An important point is that SimpleDB has been designed in the context of On-Line Transaction Processing (OLTP) SQL applications. Interaction between some code and a relational database is often said to occur either in an OLTP or in an OLAP (On-Line Analytical Processing) context. In a nutshell, OLTP corresponds to the case where the database must be read and written to support human activities in real time, like managing concurrent bank transactions, consulting customer files, processing ticket registrations or registering book borrowings. The OLAP context occurs when the data gathered in OLTP are patiently analyzed using data mining techniques, typically for reporting or strategical purposes, like accounting, budget or marketing.

An important characteristic of OLTP SQL applications (and thus of SimpleDB applications) is that they are small and involve just a few simple SQL operations over a small part of the database. In [Binnig et al., 2008], SQL code designed in an OLTP context is said to “implement use cases which execute a sequence of actions whereas each action usually reads or updates only a small set of tuples in the database”. An example of such a use case is a bookshop user that wants to buy a book. The code checks the user and book data and possibly allows and saves the buying. The documentation of Microsoft’s DBMS SQL Server 2012 ¹ describes SQL code in an OLTP context as “characterized by small, interactive transactions that generally require sub-second response times”.

In practice, SimpleDB enables writing SQL applications composed of a single Java method that implements an OLTP use case. A SimpleDB program is composed of the database’s DDL schema and of the code of the method. The database schema describes a set of tables composed of integer attributes. Each table has a primary key and the attributes can be constrained by foreign key constraints and check constraints. The code of the method can contain if-then-else blocks, while loops, assertions, return statements and integer and list (local) variable assignments, with typical operators for lists and integers. The method interacts with the database through SQL base statements whose structure is entirely described statically, and through SQL base primitives for transaction management. Statements writing in the database can throw runtime exceptions if the write operation violates the DDL schema. Such exceptions can be caught. The method receives as input parameters a JDBC connection to the database, a set of integer lists and a text scanner

¹[http://technet.microsoft.com/en-us/library/hh393556\(v=sql.110\).aspx](http://technet.microsoft.com/en-us/library/hh393556(v=sql.110).aspx)

reading integer values. The lists model any structured group of inputs transmitted to the code at method call. The scanner models the method's access to simple data from the “outside world”, like user prompt, network access, etc.

Figure 6.1 provides a sample SQL application written in SimpleDB. This SimpleDB program describes a database with two tables: one for bookshop shelves and one for the books stored in each of these shelves. The total number of books stored in a shelf is saved for each shelf. The Java method adds a set of new books to the database and updates the shelves' books counts. If a book is added to a non-existent shelf, then the shelf is itself added to the database as well. The books are inserted one by one in isolated transactions. If a transaction was successful, the code of the added book is saved in a list, which is returned at the end of the method's execution.

6.2 A Formal Definition for SimpleDB

We define now precisely—using a BNF grammar—what SimpleDB programs are. This description is made with the requirement that such programs are well-typed according to the relevant standard Java and SQL typing rules.

In the next paragraphs, the chosen notation for the BNF is standard but includes some additional meta-symbols: $\{\dots\}$ (grouping), $?$ (zero or one times), $*$ (zero or more times) and $+$ (one or more times). When a single nonterminal appears several times in a single production, subscript notation enables distinguishing between the occurrences.

6.2.1 SimpleDB Program

A SimpleDB program is composed of the SQL DDL code of the database schema and of the code of the Java method implementing an OLTP use case.

$$\langle \text{simpleDB-program} \rangle ::= \langle \text{sql-ddl} \rangle \langle \text{java-method} \rangle$$

6.2.2 Database Schema

The relational database schema is a list of table definitions. This list can be empty, in what case the program is a traditional program that works independently of any database. In the list, each table is identified by its name, contains at least one attribute and endorses exactly one primary key. Foreign keys and additional check constraints can be declared for a table. A row in a table cannot be deleted or see its primary key value modified as long as there exists at least another row in the database that references it (ON DELETE/UPDATE NO ACTION). Semantics of all the schema creation primitives conforms to the classical SQL DDL specification provided by ISO.

$$\langle \text{sql-ddl} \rangle ::= \langle \text{table} \rangle^*$$

$$\langle \text{table} \rangle ::= \text{CREATE TABLE } \langle \text{id} \rangle (\langle \text{att} \rangle^+ \langle \text{p-key} \rangle \langle \text{f-key} \rangle^* \langle \text{chk} \rangle^*);$$

$$\langle \text{att} \rangle ::= \langle \text{id} \rangle \text{ INTEGER NOT NULL ,}$$

$$\langle \text{p-key} \rangle ::= \text{CONSTRAINT } \langle \text{id} \rangle_{\text{cst}} \text{ PRIMARY KEY (} \langle \text{id} \rangle_{\text{att}} \text{)}$$

$$\langle \text{f-key} \rangle ::= \text{,CONSTRAINT } \langle \text{id} \rangle_{\text{cst}} \text{ FOREIGN KEY (} \langle \text{id} \rangle_{\text{att}} \text{) REFERENCES } \langle \text{id} \rangle_{\text{tab}} (\langle \text{id} \rangle_{\text{refid}})$$

$$\langle \text{chk} \rangle ::= \text{,CHECK (} \langle \text{id} \rangle \{ < | = | > \} \langle \text{integer} \rangle)$$

Figure 6.1: SQL and Java/JDBC code of a SimpleDB program adding books in a bookshop database.

<pre> CREATE TABLE shelf (id INTEGER NOT NULL, numberOfBooks INTEGER NOT NULL, CONSTRAINT sPK PRIMARY KEY (id), CHECK(numberOfBooks > 0)); </pre>	<pre> CREATE TABLE book (code INTEGER NOT NULL, shelfId INTEGER NOT NULL, CONSTRAINT bPK PRIMARY KEY (code), CONSTRAINT bFK FOREIGN KEY(shelfId) REFERENCES shelf (id)); </pre>
---	--


```

1 List < Integer> addBooks (Connection con, Scanner in, List< Integer> newBooks) throws SQLException {
2   int i = 0; List< Integer> addedBooks = new ArrayList< Integer> ();
3   while ( !(newBooks == null) & (i < newBooks.size()) ) {
4     int error = 0; int theShelf = in.nextInt();
5     ResultSet shelves = con.createStatement().executeQuery("SELECT id FROM shelf WHERE id=" + theShelf);
6     if ( ! shelves.next() )
7       con.createStatement().execute("INSERT INTO shelf VALUES (" + theShelf + ",1)");
8     else
9       con.createStatement().execute("UPDATE shelf SET numberOfBooks=numberOfBooks+ 1 WHERE id =" + theShelf);
10    try {
11      con.createStatement().execute("INSERT INTO book VALUES (" + newBooks.get(i).intValue() + ", " + theShelf + ")");
12    } catch (SQLException e) {
13      error = 1;
14    };
15    if (error ==0) {
16      con.commit();
17      addedBooks.add(newBooks.get(i).intValue());
18    } else
19      con.rollback ();
20    i = i + 1;
21  };
22  return addedBooks; }

```

$\langle id \rangle ::= \{a \mid \dots \mid z \mid A \mid \dots \mid Z\} \{a \mid \dots \mid z \mid A \mid \dots \mid Z \mid 0 \mid \dots \mid 9\}^*$

$\langle integer \rangle ::= -^? \{1 \mid \dots \mid 9\} \{0 \mid \dots \mid 9\}^* \mid 0\}$

6.2.3 Method Signature and Body

In SimpleDB, we consider Java methods manipulating only internal variables and parameters. Variables can only be typed as ‘int’, ‘java.util.List<java.lang.Integer>’ or ‘java.sql.ResultSet’. The method receives as input parameters a connection to the database (typed as ‘java.sql.Connection’), a scanner (typed as ‘java.util.Scanner’) and some lists of integers (typed as ‘java.util.List<java.lang.Integer>’), where two distinct list parameters cannot reference a single list object. Its return type can be either ‘void’, ‘int’ or ‘java.util.List<java.lang.Integer>’.

$\langle java-method \rangle ::= \langle type \rangle \langle id \rangle (\langle db-con \rangle, \langle inp \rangle \langle parameters \rangle) \text{ throws } SQLException$
 $\{ \langle stmt \rangle^* \}$

$\langle type \rangle ::= \text{void}$
 $\mid \text{int}$
 $\mid \text{List<Integer>}$

$\langle db-con \rangle ::= \text{Connection con}$

$\langle inp \rangle ::= \text{Scanner in}$

$\langle parameters \rangle ::= \{, \text{List<Integer>} \langle id \rangle \}^*$

The connection with the database is assumed reliable and every SQL statement, being well formed, is processed for its effect on the database. The semantics of all the Java constructs conforms to the classical Java specification and documentation. The semantics of all SQL statements conforms to the classical SQL specification provided by ISO.

A Common Statements and List Management

Common condition, loop, assertion and assignment statements, as well as common integer expressions and boolean conditions can be used. Lists can be manipulated with ‘add(int)’, ‘remove(int)’, ‘get(int)’ and ‘size()’ methods. The ‘java.util.ArrayList’ implementation of these methods is supposed to be used. A list variable can be ‘null’.

$\langle stmt \rangle ::= \text{if } (\langle cond \rangle) \{ \langle stmt \rangle_{then}^* \} \{ \text{else } \{ \langle stmt \rangle_{else}^* \} \}^?;$
 $\mid \text{while } (\langle cond \rangle) \{ \langle stmt \rangle^* \};$
 $\mid \text{assert } \langle cond \rangle ;$
 $\mid \{ \text{int} \mid \text{List<Integer>} \}^? \langle id \rangle = \langle expr \rangle ;$
 $\mid \langle id \rangle . \text{add} (\langle int-expr \rangle) ;$
 $\mid \langle id \rangle . \text{remove} (\langle int-expr \rangle) ;$
 $\mid \text{return } \langle id \rangle ;$

$\langle cond \rangle ::= \text{true}$
 $\mid \text{false}$
 $\mid (! \langle cond \rangle)$
 $\mid (\langle cond \rangle_1 \{ \& \mid \mid \} \langle cond \rangle_2)$
 $\mid (\langle int-expr \rangle_1 \{ < \mid == \mid > \} \langle int-expr \rangle_2)$
 $\mid (\langle id \rangle == \text{null})$

$$\begin{aligned} \langle expr \rangle &::= \langle int-expr \rangle \mid \langle list-expr \rangle \\ \langle int-expr \rangle &::= \langle id \rangle \\ &\mid \langle integer \rangle \\ &\mid (\langle int-expr \rangle_1 \{+ \mid -\} \langle int-expr \rangle_2) \\ &\mid (\langle id \rangle.\text{get}(\langle int-expr \rangle).\text{intValue}()) \\ &\mid (\langle id \rangle.\text{size}()) \\ \langle list-expr \rangle &::= \langle id \rangle \\ &\mid \text{null} \\ &\mid \text{new ArrayList<Integer>}() \end{aligned}$$

B Interacting with the Outside World

The scanner parameter of the method can be used to get integer data from the “outside world” (user prompt, network access, reading from a file, etc.). This interaction is assumed to always succeed, without any technical problem. We have thus the following new alternative for the $\langle stmt \rangle$ non-terminal:

$$\langle stmt \rangle ::= \{int\}^? \langle id \rangle = \text{in.nextInt}();$$

C Reading Data from the Database

Data can be read from the database using simple SQL queries. The obtained `ResultSet` can be accessed using the ‘`next()`’ and ‘`getInt(String)`’ methods. We have thus the following new terminals:

$$\begin{aligned} \langle select-query \rangle &::= \text{SELECT } \{\langle id \rangle_i, \}^* \langle id \rangle_n \text{ FROM } \langle id \rangle_{tab} \{ \text{WHERE } \langle db-cond \rangle \}^? \\ \langle db-cond \rangle &::= (\langle db-cond \rangle_1 \{ \text{AND} \mid \text{OR} \} \langle db-cond \rangle_2) \\ &\mid (\text{NOT } \langle db-cond \rangle) \\ &\mid (\langle id \rangle \{ < \mid = \mid > \} \langle db-int-expr \rangle) \\ \langle db-int-expr \rangle &::= \langle id \rangle \\ &\mid \langle integer \rangle \\ &\mid (\langle db-int-expr \rangle_1 \{+ \mid -\} \langle db-int-expr \rangle_2) \\ &\mid "+" \langle int-expr \rangle "+" \end{aligned}$$

and new alternatives for existing terminals:

$$\begin{aligned} \langle stmt \rangle &::= \{ \text{ResultSet} \}^? \langle id \rangle = \text{con.createStatement().executeQuery}(\langle select-query \rangle); \\ &\mid \langle id \rangle.\text{next}(); \\ \langle int-expr \rangle &::= \langle id \rangle_{tab}.\text{getInt}(\langle id \rangle_{att}) \\ \langle cond \rangle &::= (\langle id \rangle.\text{next}(\langle int-expr \rangle)) \end{aligned}$$

In SimpleDB, integer expressions appearing in SQL code ($\langle db-int-expr \rangle$) can be parameterized by integer constants computed dynamically by the Java method. For example, in:

```

void example (Connection con, Scanner in) throws SQLException {
    int n = 0;
    ResultSet r= con.createStatement().executeQuery("SELECT id FROM table WHERE id="+ n+ "");
    r.next();
    int p = r.getInt("id");
}

```

The character string that will be effectively sent to and processed by DBMS is:

```
SELECT id FROM table WHERE id=0
```

The parametric constants can depend on the method's inputs, like in:

```

void example (Connection con, Scanner in) throws SQLException {
    int n = in.nextInt();
    ResultSet r= con.createStatement().executeQuery("SELECT id FROM table WHERE id="+ n+ "");
}

```

D Writing Data into the Database

Data can be written into the database using simple SQL INSERT, UPDATE or DELETE statements. If the execution of such a statement provokes a violation of one of the database schema integrity constraints, the database remains unmodified by the statement, an exception is thrown within the program and the method's execution is stopped. Such exceptions should be caught using a try/catch structure.

```

<stmt> ::= con.createStatement().execute(" <db-write> ");
| try { con.createStatement().execute(" <db-write> "); }
  catch (SQLException e)
    { <stmt>* };

```

```

<db-write> ::= INSERT INTO <id> VALUES ( { <int-expr>i, }* <int-expr>n )
| UPDATE <id>tab SET <id>att = <db-int-expr> { WHERE <db-cond> }?
| DELETE FROM <id> { WHERE <db-cond> }?

```

E Transaction Management

SQL transactions are managed through the classical commit and rollback statements. We suppose that a new transaction is automatically started at the beginning of the method's execution. The first call to commit or rollback will end this transaction and then starts a new one. Any subsequent call to commit or rollback will end the current transaction and start a new one. When a commit statement is executed, it makes permanent all the changes made to the database by the program since the current transaction was started. When a rollback statement is executed, it restores the database to its state at the start of the current transaction. We suppose that all the changes made to the database since the last transaction was started are automatically committed at the end of the method's execution.

```

<stmt> ::= con.commit() ;
| con.rollback() ;

```

Chapter 7

A Relational Algorithm for Symbolic Execution

Contents

7.1	Principle	127
7.2	Alloy Version	128
7.2.1	Execution Example	128
7.2.2	Constraint Generation Rules	133
7.3	SMT-Lib Version	138
7.3.1	Execution Example	138
7.3.2	Constraint Generation Rules	144

THIS chapter details the symbolic execution algorithm that we have developed for SimpleDB programs, implementing the principle of relational testing. This algorithm comes in two versions, one based on Alloy and one on SMT-Lib. It will serve as a basis for our experimental evaluation of relational testing.

Section 7.1 introduces the main principles of our relational symbolic execution algorithm. Sections 7.2 and 7.3 detail respectively the Alloy and SMT-Lib versions of the algorithm. In both cases, we present a sample execution of the algorithm and then provide a more formal description of its behavior.

7.1 Principle

In order to evaluate our relational testing approach, we have built a symbolic execution algorithm for SimpleDB programs. In a nutshell, the proposed algorithm, receives as inputs the SimpleDB program and a single path through the Java code of this program. It produces as output a constraint system mixing classical constraints with relational constraints, according to the principle of relational testing. Each solution to this system defines inputs for the SimpleDB program that trigger the given path.

The algorithm was built in two versions, a first one that generates Alloy constraints and a second one that produces SMT-Lib constraints. The availability of these two versions will enable us to compare the performance of the Alloy solver and of the Z3 SMT solver for relational testing.

Coupling the developed symbolic execution algorithm with any existing path exploration mechanism enables generating test data for any SimpleDB program. The set of paths for which test data are computed, as well as the process used to select these paths, are thus parameters of the relational testing implementation proposed here.

The path received as input by our algorithm is supposed to be a finite path in the method's control flow graph. It defines which branches were taken at each of the encountered if statements, how many times the body of the encountered loop was executed (this number must be finite), which assertions were violated and for each encountered try/catch statement whether the catch clause was executed.

Solving the constraint system generated by the algorithm for a given path enables finding inputs triggering this path. These inputs include the content of each database table at the start of the method's execution, the value of every list received as argument by the method, and a value for the part of the input stream that is consumed during the method's execution. If the constraint system produced for a given path has no solution, this means that the path is infeasible. As the produced constraints are written in a logic that is not decidable in general, it can happen that for a given path the solver may neither be able to find a solution for the generated constraint system, nor be able to establish that such a solution does not exist.

In practice, the algorithm works as follows. First, symbolic execution generates constraints over the symbols representing the initial values of the database tables. These constraints state that, initially, each table contains data that conform to the database schema. Then, symbolic execution analyses one by one the method's statements in the order specified by the path. Each time a statement sets or changes the value of a method variable or database table, symbolic execution generates constraints over the symbols representing the new value. These constraints define how the new value can be computed from the values of the database tables and program variables before the statement's execution. Moreover, every time the value of a database table is changed, constraints are also added to state that the new value satisfies the database schema. Finally, every time an if, while, assert or try/catch statement is encountered, symbolic execution generates an additional constraint over the symbols such that when the program is executed with respect to values satisfying this constraint, the execution is guaranteed to take the considered path.

A detailed description of both the Alloy and SMT-Lib versions of the algorithm are respectively presented in the two next sections.

7.2 Alloy Version

7.2.1 Execution Example

In this subsection, we illustrate the execution of the Alloy version of the algorithm over the sample SimpleDB program of Figure 6.1 (page 121). We detail each step of the symbolic execution process over the path where the while loop is executed once, the then branch of both the if statements are taken, and the catch clause of the try/catch is not executed (lines 1-7, 10-11, 15-17, 3 and 22).

At each step, we present the rules used by our algorithm to generate the corresponding Alloy symbols and constraints. All the rules that are part of the complete set of rules defining our symbolic execution algorithm for SimpleDB are presented during one of these steps, and/or described formally in a set of tables detailed in the next subsection.

The first step executed by our algorithm is to generate Alloy symbols and constraints for the SQL DDL code of the database schema. For the database schema described in Figure 6.1, the generated Alloy code is detailed in the frame below. For the reader's convenience, the corresponding SimpleDB code will be reminded as a comment in the Alloy code.

For each table, an Alloy type is defined using the Alloy object-oriented notation (the lines prefixed by (1) in the Alloy code below), representing the set of all the possible tuples that could appear in the table. Every symbol representing a set of rows from the table will thus be a subset of this type. Then, a symbol is created to represent the initial content of every table (2). Finally, constraints are generated to enforce on this content the primary key (3) as well as all the foreign keys (4) and check constraints (5) defined in the schema. Note that the original SQL table names, as well as the original Java variable names, are used as Alloy symbols, suffixed by the natural number 1 (e.g. *shelf1* or *book1* in the Alloy code below), which represents the fact that the current symbols represent the initial values of the represented tables or variables. Subsequent values of a same table or variable will be represented by the same symbol suffixed with successive numbers.

```
// CREATE TABLE shelf (  
// id INTEGER NOT NULL,  
// numberOfBooks INTEGER NOT NULL,  
// CONSTRAINT sPK PRIMARY KEY (id),  
// CHECK(numberOfBooks > 0));  
  
// CREATE TABLE book (  
// code INTEGER NOT NULL,  
// shelfId INTEGER NOT NULL,  
// CONSTRAINT bPK PRIMARY KEY (code),  
// CONSTRAINT bFK FOREIGN KEY(shelfId) REFERENCES shelf (id));  
  
module bookShop // Name of the Alloy constraint set  
  
// ----- TABLE shelf -----  
// New type definition  
(1) sig shelf {id : Int,numberOfBooks : Int}  
// Helper code  
pred equalshelf[a:shelf , b: shelf] // Equality (macro) predicate over tuples
```

```

    { a.id = b.id && a.numberOfBooks = b.numberOfBooks }
fact{all disj a,b: shelf | !equalshelf[a,b]} // Distinct elements represent distinct tuples

// Initial table content
(2) sig shelf1 in shelf {}

// Integrity constraints
(3) fact{all disj a,b: shelf1 | !((a.id = b.id))}
(5) fact{all a: shelf | a.numberOfBooks > 0}
// -----

// ----- TABLE book -----
// New type definition
(1) sig book{code : Int,shelfId : Int}
// Helper code
pred equalbook[a:book,b: book] // Equality (macro) predicate over tuples
    {a.code = b.code && a.shelfId = b.shelfId}
fact{all disj a,b: book | !equalbook[a,b]} // Distinct elements represent distinct tuples

// Initial table content
(2) sig book1 in book {}

// Integrity constraints
(3) fact{all disj a,b: book1 | !((a.code = b.code))}
(4) fact{all a: book1 | one b:shelf1 | a.shelfId = b.id}
// -----

```

The second step executed by our algorithm is to generate another Alloy type for lists of integers (1), and generic Alloy macro predicates and functions for their add/remove/get/size methods (2). Lists are modeled in Alloy as an interval of indexes from 0 to a natural number, where each index i is mapped to an integer value representing the i^{th} element of the list. The ‘null’ value is represented by the Null singleton type (3).

```

// New types
(1) sig List { index: set Int,elems: index  $\rightarrow$  one Int }
(3) one sig Null {}

// Helper code
// Macro function returning the smallest element of a set of integers
fun head[l: set Int] : Int { { e:l | (all p: l | !(e=p))  $\Leftrightarrow$  (e< p)) } }
// Macro function removing the smallest element from a set of integers
fun tail[l: set Int] : set Int { l - { e:l | (all p: l | !(e=p))  $\Leftrightarrow$  (e< p)) } }
// Macro predicate verifying if a set of integer is a continuous interval
pred isSuccessive[l: set Int] { all i:tail[l] | (one j:l | i = 1.add[j]) }
// Constraint stating that non-empty list indexes are
// continuous intervals of the form [0,n] (n >= 0)
fact { all l:List | (#l.index=0 || head[l.index]=0 && isSuccessive[l.index]) }
// Macro predicate verifying if the list is null
pred isNull[l: List + Null] { l=NULL }
// Macro predicate verifying if the list is empty
pred isEmpty[l: List] { #l.index=0 }

```

```

// Macro functions/predicates for methods add, remove, get and size
(2) pred add[l: List, e: Int, newList: List] // is true iff l.add(e) = newList
    { newList.index = l.index + #l.index
      && (all i: l.index | newList.elems[i] = l.elems[i]) && newList.elems[#l.index] = e }
(2) pred remove[l: List, e: Int, newList: List] // is true iff l.remove(e) = newList
    { size [l] ≥ e && newList.index = l.index - (#l.index).sub[1]
      && (all i: newList.index | (i < e ⇔ newList.elems[i] = l.elems[i])
      && (i ≥ e ⇔ newList.elems[i] = l.elems[i.add[1]])) }
(2) fun get[l: List, i: Int] : Int { l.elems[i] }
(2) fun size[l: List] : Int { #l.index }

```

The third step executed by our algorithm is to define symbols (typed as members of the union of lists and the null value) for the initial content of each list parameter of the method. For the example method considered in this section, the following code is generated:

```

// INPUT PARAMETER: List<Integer> newBooks
one sig newbooks1 in List + Null {}

```

The algorithm can then proceed with the symbolic execution of the method. It follows the path received as input and considers all statements one by one. In the case of our example, the two first statements to be executed are assignments. Symbolic execution for assignment creates a new symbol of the correct type to represent the new value of the assigned variable (1) and generates constraints to specify that this new symbol contains the value computed by evaluating the expression on the right of the '=' symbol (2). In the particular case where a list variable is assigned to a different list variable, the shared content of the two variables is represented by a single symbol.

```

// int i = 0;
(1) one sig i1 in Int {}
(2) fact { i1 = 0 }

```

```

// List<Integer> addedBooks = new ArrayList<Integer>();
(1) one sig addedbooks1 in List {}
(2) fact { isEmpty[addedbooks1] }

```

The next statement in the path is a while statement. As the path specifies that the loop body must be executed, a constraint is generated to specify that the loop condition at this point of time should be true (1). As the 'size()' method cannot be called on a null object without causing a runtime error, a constraint is automatically added to ensure that the current value of the 'newBooks' variable of the method is not null (2).

```

// ENTERING LOOP: while ( !(newBooks == null) & (i < newBooks.size()) )
(1) fact { i1 < size [newbooks1] }
(2) fact { !isNull [newbooks1] }

```

Then the algorithm proceeds with symbolic execution of the statements in the loop body, as specified within the input path. The first statement is an assignment statement:

```

// int error = 0;
one sig error1 in Int {}
fact { error1 = 0 }

```

Symbolic execution for use of the input scanner simply creates a new symbol to represent the scanned value:

```
// int theShelf = in.nextInt();
one sig theshelf1 in Int { }
```

Symbolic execution for select statements creates a new symbol (1) to represent the content of the ResultSet variable. This symbol is typed as a set of rows from the table on which the select was executed. A constraint is then generated (2) to specify that a row is part of the ResultSet if and only if it is part of the current content of the table on which the select query is executed and that it enforces the WHERE condition of the select query:

```
// ResultSet shelves
// = con.createStatement().executeQuery("SELECT id FROM shelf WHERE id="+theShelf);
(1) sig shelves1 in shelf { }
(2) fact { all e:shelf | (e in shelf1 && (e.id = theshelf1))  $\Leftrightarrow$  e in shelves1 }
```

Symbolic execution for select also creates a symbol to represent the specific order in which the rows were returned by the select query. This symbol (1) represents an interval of indexes from 0 to the number of rows returned by the select statement (2), where each index is mapped to one of the returned rows (3):

```
// ResultSet shelves
(1) sig shelves1indexes in Int { mapshelves1: one shelves1 }
(2) fact { (#shelves1indexes=0 ||
  (head[shelves1indexes]=0 && isSuccessive[shelves1indexes]))&& #shelves1indexes=#shelves1 }
(3) fact { all disj a,b: shelves1indexes | !(a.mapshelves1=b.mapshelves1) }
```

As the path specifies that the then branch of the if statement must be executed this time, a constraint is generated to specify that the condition of the if should be true, i.e. that `shelves.next()` should return false.

For each ResultSet object, the algorithm records the number of times the ‘`next()`’ method has been called on this object. This value represents the index increased by one of the row pointed by the cursor of the ResultSet at the current execution state of the path. When the boolean value returned by the ‘`next()`’ method is used in an if or while condition, this value states if the number of rows in the ResultSet is greater or equal to the number of times the ‘`next()`’ method has been called so far on this ResultSet. In this case, `shelves.next()` will return true if the ResultSet `shelves` contains less than one row (as `shelves.next()` has been called once on the ResultSet):

```
// TAKING THEN BRANCH OF: if ( ! shelves.next() )
fact{!(#shelves1indexes ≥ 1)}
```

Symbolic execution for insert creates a new symbol (1) for the new content of the table on which the insert statement is executed. Then a constraint is generated stating that this new variable can be obtained by adding one row with the correct attribute values to the old one (2). Finally, constraints are added to specify that no constraint was violated during insert. In this case, a constraint is added to state that there should not be any row in the previous content of the table whose primary key value is the same as in the row inserted by the statement (3):

```
// con.createStatement().execute("INSERT INTO shelf VALUES (" + theShelf + ",1)");
(1) sig shelf2 in shelf { }
(2) fact { one e:shelf | shelf2=shelf1+ e && e.numberOfBooks=1 && e.id=theshelf1 }
(3) fact { no e:shelf1 | e.id=theshelf1 }
```

Subsequently, as in our example the path specifies that the catch block of the try/catch statement must not be executed. The second insert statement is thus executed in the same way as the first one (1) and a relational constraint (2) is added to state that the inserted book should reference an existing row in the shelf table:

```
// NOT TAKING THE CATCH BRANCH OF:
// try { con.createStatement().execute(
// "INSERT INTO book VALUES (" + newBooks.get(i).intValue() + "," + theShelf + ")");
// } catch (SQLException e) {
(1) sig book2 in book {}
(1) fact{one e:book | book2=book1 + e && e.shelfId=theshelf1 && e.code=get[newbooks1,i1]}
(1) fact {no e:book1|e.code=get[newbooks1,i1]}
(2) fact{one e:shelf2|e.id=theshelf1}
```

As the path specifies that the then branch of the if statement must be executed this time, a constraint is generated to specify that the if condition should be true:

```
// TAKING THEN BRANCH OF: if (error==0)
fact { error1 = 0 }
```

Symbolic execution for Commit statements simply does nothing. Symbolic execution for rollback statements tells the algorithm to represent the current content of each database table using the symbols that were representing the content of the table just before the last start of a new transaction (saved by the algorithm at the beginning of the method execution and after each call to commit or abort).

Symbolic execution for Add and Remove statements creates a new symbol (1) to represent the new content of each variable which is currently referencing the list object modified by the statement. A constraint is added (2) to relate the old and new object referenced by these variables using the add/remove macro predicates defined earlier:

```
// addedBooks.add(newBooks.get(i).intValue());
(1) one sig addedBooks2 in List {}
(2) fact { add[addedbooks1,newbooks1-sub[1],addedBooks2] }
```

The assignment statement is then symbolically executed:

```
// i = i + 1;
one sig i2 in Int {}
fact { i2 = i1.add[1] }
```

As the path specifies that the loop body must not be executed any more, a constraint is generated to specify that, at this point in time, the loop condition should be false:

```
// ESCAPING LOOP: while ( !(newBooks == null) & (i < newBooks.size()) )
fact { !( (i2 != null && (i2 < size[newbooks1])) ) }
```

As a return statement is met, the algorithm stops and returns the generated Alloy constraint model. The Alloy solver can now be asked to find a valuation for the defined symbols satisfying the constraints (and violating an always false assertion). As the algorithm records what symbols represent the initial, respectively final, values of a variable or table, the input values of the method (for the considered path) can easily be extracted from the solution to the constraint system. For our example, the set of 25 constraints was solved by Alloy 4.2 in 340ms (1.8 GHz Intel Core i5, 8GB Ram) and the test data (we have included the method's outputs as well, as they constitute useful data to identify code defects) that

were obtained from the solution to the constraint system are summarized in the following tables:

Inputs			Outputs										
Name	Symbol(s)	Value	Name	Symbolic(s)	Value								
TABLE shelf	CONTENT: <i>shelf1</i> , ATTRIBUTES: <i>id</i> <i>numberOfBooks</i>	<table><tr><td>id</td><td>n.Books</td></tr><tr><td colspan="2">empty</td></tr></table>	id	n.Books	empty		TABLE shelf	CONTENT: <i>shelf1</i> , ATTRIBUTES: <i>id</i> <i>numberOfBooks</i>	<table><tr><td>id</td><td>n.Books</td></tr><tr><td>7</td><td>1</td></tr></table>	id	n.Books	7	1
id	n.Books												
empty													
id	n.Books												
7	1												
TABLE book	CONTENT: <i>book1</i> ATTRIBUTES: <i>code</i> <i>shelfId</i>	<table><tr><td>code</td><td>s.Id</td></tr><tr><td colspan="2">empty</td></tr></table>	code	s.Id	empty		TABLE book	CONTENT: <i>book1</i> ATTRIBUTES: <i>code</i> <i>shelfId</i>	<table><tr><td>code</td><td>s.Id</td></tr><tr><td>-3</td><td>7</td></tr></table>	code	s.Id	-3	7
code	s.Id												
empty													
code	s.Id												
-3	7												
newBooks	<i>newbooks1</i>	$[-3]$	newBooks	<i>newbooks1</i>	$[-3]$								
in.nextInt()	<i>theshelf1</i>	$[7]$	addedBooks	<i>addedbooks1</i>	$[-3]$								

7.2.2 Constraint Generation Rules

For sake of completeness, the following tables define the constraint generation rules used by our algorithm in case of an Insert (Table 7.1), Update (Table 7.2) and Delete (Table 7.3) statements. Table 7.4 explains the abbreviations used in the previous tables. Table 7.5 describes the full set of rules used by the algorithm to translate between SimpleDB and Alloy expressions and conditions.

In each of these tables, any declaration of a new symbol leverages a generator providing a fresh symbol identifier, i.e. which has still not been used in the Alloy code generated so far. This is denoted by:

sig freshAlloySym **in** $\langle id \rangle$ $\{ \}$

All the subsequent references to **freshAlloySym** in the table represent this newly declared symbol. The generated fresh symbols are named according to the naming rule detailed along the example given in the previous subsection.

Finally, assertions are handled as if statements. For example :

assert $x == 0$;

is handled as:

if ($x == 0$)

End of computation with AssertionError

Table 7.1: Alloy constraints generation rules for INSERT statements

INSERT INTO $\langle id \rangle$ VALUES ($\langle int-expr \rangle_1$, ... , $\langle int-expr \rangle_i$, ... , $\langle int-expr \rangle_n$)
<pre> if (no exception thrown in path for this INSERT) { sig freshAlloySym in $\langle id \rangle$ {} fact { one e:$\langle id \rangle$ freshAlloySym=alloyName+e && e.att_* = alloyOf($\langle int-expr \rangle_*$) } // Inserted primary key value does not already exist fact { no e:alloyName e.pk =alloyOf($\langle int-expr \rangle_{pk^{pos}}$) } // Inserted values constrained by a foreign key reference existing rows fact { one e:fk_*^{tab} e.fk_*^{pk} = alloyOf($\langle int-expr \rangle_{fk^{pos}}$) } } else { // Logical disjunction between every possible constraint // violation given the database schema and this insert: // The inserted primary key value already exists in the table: fact { one e:alloyName e.pk =alloyOf($\langle int-expr \rangle_{pk^{pos}}$) } // ith inserted foreign key value does not reference a row: fact { no e:fk_i^{tab} e.fk_i^{pk} = alloyOf($\langle int-expr \rangle_{fk_i^{pos}}$) } // An inserted attribute can violate a check constraint: fact {!($\langle int-expr \rangle_{co^{pos}}$ co^{right}) } } </pre>

Table 7.2: Alloy constraints generation for UPDATE statements

UPDATE $\langle id \rangle$ SET $\langle id \rangle_{att} = \langle db-int-expr \rangle$ WHERE $\langle db-cond \rangle$
<pre> if (no exception thrown in path for this UPDATE) { sig freshAlloySym in $\langle id \rangle$ {} fact { all e:alloyName (alloyOf($\langle db-cond \rangle$, $\langle id \rangle$, e) && (one y:freshAlloySym y.att_{*-$\{\langle id \rangle_{att}\}$} = e.att_{*-$\{\langle id \rangle_{att}\}$} && y.$\langle id \rangle_{att}$ = alloyOf($\langle db-int-expr \rangle$, $\langle id \rangle$, e))) (! (alloyOf($\langle db-cond \rangle$, $\langle id \rangle$, e)) && (one y:freshAlloySym equal$\langle id \rangle$[y,e]))) } fact { all y:freshAlloySym one e:alloyName (alloyOf($\langle db-cond \rangle$, $\langle id \rangle$, e) && y.att_{*-$\{\langle id \rangle_{att}\}$} = e.att_{*-$\{\langle id \rangle_{att}\}$} && y.$\langle id \rangle_{att}$ = alloyOf($\langle db-int-expr \rangle$, $\langle id \rangle$, e))) (! (alloyOf($\langle db-cond \rangle$, $\langle id \rangle$, e)) && (one y:freshAlloySym equal$\langle id \rangle$[y,e]))) } // Primary key is not violated fact { all disj a,b:freshAlloySym !(a.pk = b.pk) } // If $\langle id \rangle_{att} = fk_i$, updated rows should still reference a row fact { all a:freshAlloySym one b:fk_i^{tab} a.$\langle id \rangle_{att}$ = b.fk_i^{pk} } // If $\langle id \rangle_{att} = pk$, none of the updated rows should have been referenced by another row fact { all e:alloyName no f:ifk_*^{tab} (alloyOf($\langle db-cond \rangle$, $\langle id \rangle$, e) && e.$\langle id \rangle_{att}$ = f.ifk_*^{att}) } } else { // Logical disjunction between every possible constraint // violation given the database schema and this update: // Update on primary key leads to duplicate primary keys: fact { some disj a,b:alloyName ((alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, a) && alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, b) && (alloyOf($\langle db-int-expr \rangle$, $\langle id \rangle$, a) = alloyOf($\langle db-int-expr \rangle$, $\langle id \rangle$, b))) (! (alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, a) && alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, b) && (a.$\langle id \rangle_{att}$ = alloyOf($\langle db-int-expr \rangle$, $\langle id \rangle$, b))))) } // Trying to update the primary key of a referenced row: fact { some a:alloyName alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, a) && (some ifk_j^{tab} (b.ifk_j^{att} = a.$\langle id \rangle_{att}$)) } // Update on foreign key lets row without row to reference: fact { some a:alloyName alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, a) && (no b:fk_i^{tab} b.fk_i^{pk} = alloyOf($\langle db-int-expr \rangle$, $\langle id \rangle$, a)) } // An inserted attribute violates an arithmetic constraint: fact { some a:alloyName alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, a) && !(alloyOf($\langle db-int-expr \rangle$, $\langle id \rangle$, a))co_i^{right}) } } </pre>

Table 7.3: Alloy constraints generation for DELETE statements

DELETE FROM $\langle id \rangle$ WHERE $\langle db-cond \rangle$
<pre> if (no exception thrown in path for this DELETE) { sig freshAlloySym in $\langle id \rangle$ {} fact {freshAlloySym = alloyName - {e:alloyName alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, e)}} // Not trying to delete a referenced row fact {all e:alloyName no f:ifk$_j^{tab}$ (alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, e) && e.pk = f.ifk$_j^{att}$)} } else { // Logical disjunction between every possible constraint // violation given the database schema and this update: // Trying to delete a referenced row: fact {some e:alloyName alloyOf($\langle db-cond \rangle$, $\langle id \rangle_{tab}$, e) && (some f:ifk$_j^{tab}$ e.pk = f.ifk$_j^{att}$)} } } </pre>

Table 7.4: Alloy Translation Abbreviation List

Abbreviation	Meaning
alloyName $_{subscript}^{superscript}$	if ($\langle id \rangle_{subscript}^{superscript}$ refers to a database table name) then The symbol that represents the current content of table $\langle id \rangle_{subscript}^{superscript}$ else The symbol that represents the current content of the Java variable $\langle id \rangle_{subscript}^{superscript}$
numberOfCallsToNext $_{sub}$	Number of call made to the next() method of the ResultSet object in variable $\langle id \rangle_{sub}$
att $_i$	Name of the i_{th} attribute in the list of attributes of table $\langle id \rangle$
pk $_{subscript}^{superscript}$	Name of the primary key attribute of table $\langle id \rangle_{subscript}^{superscript}$
pk ^{pos}	Position of primary key in the list of attributes of table $\langle id \rangle$
fk $_i^{tab}$	Name of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id \rangle$
fk $_i^{pk}$	Name of the primary key attribute of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id \rangle$
fk $_i^{pos}$	Position of the foreign key attribute, declared by the i_{th} foreign key in the list of table $\langle id \rangle$, in the list of attributes of table $\langle id \rangle$
fk $_i$	Name of the foreign key attribute declared by the i_{th} foreign key in the list of table $\langle id \rangle$
ifk $_i^{tab}$	Name of the table where is declared the i_{th} foreign key referencing table $\langle id \rangle$ in the whole schema
ifk $_i^{att}$	Name of the foreign key attribute declared by the i_{th} foreign key referencing table $\langle id \rangle$ in the schema
co $_i^{pos}$	Position of the attribute constrained by the i_{th} check constraint declared in table $\langle id \rangle$
co $_i^{right}$	Right part of the i_{th} check constraint declared in table $\langle id \rangle$ (i.e. right part of “a>0” is “>0”)
xx* means “for each xx $_i$ ” and xx*-{y} means “for each xx $_i$ except from y”	

Table 7.5: Translation of SimpleDB Expressions and Conditions into Alloy

Parameters	alloyOf(Paramaters)
$\langle id \rangle$	alloyName
$z \in \mathbb{Z}$	z
$\langle \langle int-expr \rangle_1 \{+ - \} \langle int-expr \rangle_2 \rangle$	$(\text{alloyOf}(\langle int-expr \rangle_1), \{ \text{add} \mid \text{sub} \} [\text{alloyOf}(\langle int-expr \rangle_2)])$
$\langle - \langle int-expr \rangle \rangle$	$(- (\text{alloyOf}(\langle int-expr \rangle)))$
$\langle id \rangle . \text{get}(\langle int-expr \rangle)$	$\text{get}[\text{alloyName}, \text{alloyOf}(\langle int-expr \rangle)]$
$\langle id \rangle . \text{size}()$	$\text{size}[\text{alloyName}]$
$\langle id \rangle_{tab} . \text{getInt}(\langle id \rangle_{att})$	$\text{numberOfCallsToNext}_{tab} . \text{map}[\text{alloyName}_{tab}[\langle id \rangle_{att}]]$
$\langle id \rangle = \text{null};$	$\text{fact} \{ \# \text{alloyName}_{tab} . \text{indexes} > = \text{numberOfCallsToNext}_{tab} \}$
$\langle id \rangle = \text{new ArrayList} < \text{Integer} > ();$	$\text{fact} \{ \text{isEmpty}[\text{alloyName}] \}$
true	$(0=0)$
false	$(0=1)$
$\langle \langle cond \rangle_1 \{ \& \} \langle cond \rangle_2 \rangle$	$(\text{alloyOf}(\langle cond \rangle_1) \{ \& \& \} \text{alloyOf}(\langle cond \rangle_2))$
$\langle ! \langle cond \rangle \rangle$	$!(\text{alloyOf}(\langle cond \rangle))$
$\langle \langle int-expr \rangle_1 \{ < = > \} \langle int-expr \rangle_2 \rangle$	$((\text{alloyOf}(\langle int-expr \rangle_1)) \{ < = > \} (\text{alloyOf}(\langle int-expr \rangle_2)))$
$\langle \langle id \rangle == \text{null} \rangle$	$\text{isNull}[\text{alloyName}]$
$\langle id \rangle . \text{next}()$	$(\# \text{alloyName} . \text{indexes} > = \text{numberOfCallsToNext})$
$\langle \langle db-cond \rangle_1 \{ \text{AND} \text{OR} \} \langle db-cond \rangle_2 \rangle, \text{row}$	$(\text{alloyOf}(\langle db-cond \rangle_1, \text{row}) \{ \& \& \} \text{alloyOf}(\langle db-cond \rangle_2, \text{row}))$
$\langle \text{NOT} \langle db-cond \rangle \rangle, \text{row}$	$!(\text{alloyOf}(\langle db-cond \rangle, \text{row}))$
$\langle \langle id \rangle \{ < = > \} \langle db-int-expr \rangle \rangle, \text{row}$	$(\text{row} . \langle id \rangle \{ < = > \} (\text{alloyOf}(\langle db-int-expr \rangle, \text{row})))$
$z \in \mathbb{Z}, \text{row}$	z
$\langle id \rangle, \text{row}$	$\text{row} . \langle id \rangle$
$\langle \langle db-int-expr \rangle_1 \{+ - \} \langle db-int-expr \rangle_2 \rangle, \text{row}$	$(\text{alloyOf}(\langle db-int-expr \rangle_1, \text{row}), \{ \text{add} \mid \text{sub} \} [\text{alloyOf}(\langle db-int-expr \rangle_2, \text{row})])$
$\langle - \langle db-int-expr \rangle \rangle, \text{row}$	$(- (\text{alloyOf}(\langle db-int-expr \rangle, \text{row})))$
$\langle + \langle int-expr \rangle + \rangle, \text{row}$	$\text{alloyOf}(\langle int-expr \rangle)$

7.3 SMT-Lib Version

7.3.1 Execution Example

In this subsection, we illustrate the execution of the SMT-Lib version of the algorithm over the sample SimpleDB program of Figure 6.1 (page 121). We detail each step of the symbolic execution process over the path where the while loop is executed once, the else branch of both the if statements is taken, and the catch clause of the try/catch is executed (lines 1-6, 8-15, 18-21, 3 and 22).

At each step, we present the rules used by our algorithm to generate the corresponding SMT-Lib symbols and constraints. All the rules that are part of the complete set of rules defining our symbolic execution algorithm for SimpleDB are presented during one of these steps, and/or described formally in a set of tables available at the end of this section.

The first step executed by our algorithm is to generate SMTv2 symbols and constraints for the SQL DDL code of the database schema. For the database schema described in Figure 6.1, the generated SMTv2 code is detailed in the frame below. For the reader's convenience, the corresponding SimpleDB code will be reminded as a comment in the SMT-Lib code.

First of all, the algorithm generates new symbol types for the kind of objects stored in each table defined by the schema (the lines prefixed by (0) in the SMTv2 code below). It will then generate symbols and constraints describing the input content of each of these tables. The modeling used is inspired by the one proposed in [El Ghazi and Taghdiri, 2011] for relational types. First, a symbol is created (1) to represent the initial set of objects in each table. Typed as a boolean function, it returns true for each object present in the input content of the table. Symbols typed as integer functions are then generated (2) to associate to each object in the table one of its attribute values. Finally, constraints are generated to enforce on this input content all the check constraints (3), primary key constraints (4), and foreign key constraints (5) defined in the schema.

Note that the original SQL table and attribute names, as well as the original Java variable names, are used as SMTv2 symbols, suffixed by the natural number 1 (e.g. *book1* or *id1* in the SMTv2 code below), which represents the fact that the current symbols represent the initial values of the represented tables, attributes or variables. Subsequent values of a same table, attribute or variable will be represented by the same symbol suffixed with successive numbers.

```
; CREATE TABLE book (  
; code INTEGER NOT NULL,  
; shelfId INTEGER NOT NULL,  
; CONSTRAINT bPK PRIMARY KEY (code),  
; CONSTRAINT bFK FOREIGN KEY(shelfId) REFERENCES shelf (id));  
  
; CREATE TABLE shelf (  
; id INTEGER NOT NULL,  
; numberOfBooks INTEGER NOT NULL,  
; CONSTRAINT sPK PRIMARY KEY (id),  
; CHECK(numberOfBooks > 0));
```

```

; New types for tables
(0) (declare-sort book)
(0) (declare-sort shelf)
; Input content of table Book
(1) (declare-fun book1 (book) Bool)
(2) (declare-fun shelfId1 (book) Int)
(2) (declare-fun code1 (book) Int)
(4) (assert (forall ((a book) (b book))
    ( $\Rightarrow$  (and (and (book1 a) (book1 b)) (= (code1 a) (code1 b))) (= a b))))

; Input content of table Shelf
(1) (declare-fun shelf1 (shelf) Bool)
(2) (declare-fun numberOfBooks1 (shelf) Int)
(2) (declare-fun id1 (shelf) Int)
(3) (assert (forall ((a shelf)) ( $>$  (numberOfBooks1 a) 0)))
(4) (assert (forall ((a shelf) (b shelf))
    ( $\Rightarrow$  (and (and (shelf1 a) (shelf1 b)) (= (id1 a) (id1 b))) (= a b))))

; Foreign keys
(5) (assert (forall ((a book))
    ( $\Rightarrow$  (book1 a) (exists ((b shelf)) (and (shelf1 b) (= (shelfId1 a) (id1 b)))))))

```

The second step executed by our algorithm is to define a new SMTv2 symbol type (called BoundedList) for lists of integers. All the symbols that will be subsequently generated to represent the value of a variable typed as a Java list will be part of this new SMTv2 type. A BoundedList symbol represents a record composed of three fields: the *isNull* field is typed as boolean, the *size* field is typed as integer and the *elements* field is typed as array of integers. If the *isNull* field is true, then the symbol represents the Java `null` value. Otherwise, the field *size* represents the size of the list, and the field *elements* represents an array whose indexes 0 to (*size* - 1) contain the elements of the list in the right order.

```

(declare-datatypes ()
  ((BoundedList (mk-bounded-list (isNull Bool) (size Int) (elements (Array Int Int))))))

```

It should be noted that the BoundedList symbol type is defined using an algebraic datatype declaration, where "mk-bounded-list" will be the ad-hoc constructor for list objects. If datatype declarations are handled by several SMT solvers, the related syntax has not been standardised in SMT-Lib. In this work, we use the datatype notation available in the Microsoft Z3 SMT solver.

The third step executed by our algorithm is to define symbols (typed as BoundedList) for the initial content of each list parameter of the method. For the example method considered in this section, the following code is generated:

```

; INPUT PARAMETER: List<Integer> newBooks
(declare-const newbooks1 BoundedList)
(assert ( $\Rightarrow$  (not (isNull newbooks1)) ( $\geq$  (size newbooks1) 0)))

```

It should be noted that the second constraint enforces a general property of BoundedList objects, described earlier. However, it will be enforced one by one for each BoundedList object. This is an optimisation compared to using a more general constraint, quantified over the set of all possible BoundedList objects. In order to solve such a constraint, the

solver would indeed be required to instantiate correctly the quantifier by itself, making the constraint more costly to solve.

The algorithm can then proceed with the symbolic execution of the method. It follows the path received as input and considers all the statements one by one. In the case of our example, the two first statements to be executed are assignments. Symbolic execution for assignment creates a new symbol of the correct type to represent the new value of the assigned variable (1) and generates constraints to specify that this new symbol contains the value computed by evaluating the expression on the right of the '=' symbol (2). In the particular case where a list variable is assigned to a different list variable, the shared content of the two variables is represented by a single symbol.

```
; int i = 0;
(1) (declare-const i1 Int)
(2) (assert (= i1 0))
```

```
; List<Integer> addedBooks = new ArrayList<Integer>();
(1) (declare-const addedbooks1 BoundedList)
(2) (assert (not (isNull addedbooks1)))
(2) (assert (= (size addedbooks1) 0))
```

The next statement in the path is a while statement. As the path specifies that the loop body must be executed, a constraint is generated to specify that the loop condition at this point of time should be true (1). As the 'size()' method cannot be called on a null object without causing a runtime error, a constraint is automatically added to ensure that the current value of the 'newBooks' variable of the method is not null (2).

```
; ENTERING LOOP: while ( !(newBooks == null) & (i < newBooks.size()) )
(1) (assert (and (not (isNull newbooks1)) (< i1 (size newbooks1))))
(2) (assert (not (isNull newbooks1)))
```

Then the algorithm proceeds with symbolic execution of the statements in the loop body, as specified within the input path. The first statement is an assignment statement:

```
; int error = 0;
(declare-const error1 Int)
(assert (= error1 0))
```

Symbolic execution for use of the input scanner simply creates a new symbol to represent the scanned value:

```
; int theShelf = in.nextInt();
(declare-const theshelf1 Int)
```

Symbolic execution for select statements creates new symbols to represent the content of the ResultSet variable. A first symbol (1) describes the number of rows returned by the select query. These rows are available through a second symbol (2) which is a function that returns them in the order in which they are returned by the ResultSet: (*shelves1List* 0) will be the first returned row, (*shelves1List* 1) the second one and so on.

```
; ResultSet shelves
(1) (declare-const shelves1Size Int)
(2) (declare-fun shelves1List (Int) shelf)
```

The modeling proposed in [El Ghazi and Taghdiri, 2011] for constraining the content and cardinality of relations is then used to specify that a row is part of the ResultSet if and only if it is part of the current content of the table on which the select query is executed and that it enforces the WHERE condition of the select query. In practice, new constraints are added (1) to define a function *shelves1InvertedList* which is the inverse of *shelves1List*. This function is used (1) to ensure that *shelves1List* defines a one to one correspondence between the integers $0 \leq i \leq shelves1Size$ and the elements in the ResultSet. Helper code (2) is also added to ensure a proper (pattern-based) instantiation of the universal quantifiers by the solver.

```

; = con.createStatement().executeQuery("SELECT id FROM shelf WHERE id="+theShelf);
(1) (declare-fun shelves1InvertedList (shelf) Int)
(2) (declare-fun shelves1Trigger (Int) Bool)
(1) (assert (and ( $\geq shelves1Size$  0)
    ( $\Rightarrow$  ( $= shelves1Size$  0)
    (forall ((c shelf)) (not (and (shelf1 c) ( $=$  (id1 c) theshelf1 ))))))))
(1) (assert (forall ((c shelf))
    ( $\Rightarrow$  (and (shelf1 c) ( $=$  (id1 c) theshelf1 ))
    (and ( $\geq$  (shelves1InvertedList c) 0) ( $<$  (shelves1InvertedList c) shelves1Size))))))
(1) (assert (forall ((c shelf))
    ( $\Rightarrow$  (and (shelf1 c) ( $=$  (id1 c) theshelf1 ))
    ( $=$  c (shelves1List (shelves1InvertedList c))))))
(1) (assert (forall ((i Int)
    ( $\Rightarrow$  (and ( $\geq$  i 0) ( $<$  i shelves1Size))
    ( $=$  i (shelves1InvertedList (shelves1List i))))))
(1) (assert (forall ((i Int)
    (! ( $\Rightarrow$  (and ( $\geq$  i 0) ( $<$  i shelves1Size))
    (and (shelf1 (shelves1List i)) ( $=$  (id1 (shelves1List i)) theshelf1 ))
    (pattern (shelves1Trigger i))))))
(2) (assert ( $\Rightarrow$  ( $\geq$  0 shelves1Size) (shelves1Trigger 1)))
(2) (assert (forall ((i Int)
    (! ( $\Rightarrow$  (and ( $\geq$  i 0) ( $<$  i shelves1Size))
    (shelves1Trigger (+ i 1)))
    (pattern (shelves1Trigger i))))))

```

As the path specifies that the else branch of the if statement must be executed this time, a constraint is generated to specify that the condition of the if should be false, i.e. that *shelves.next()* should return true.

For each ResultSet object, the algorithm records the number of times the *next()* method has been called on this object. This value represents the index increased by one of the row pointed by the cursor of the ResultSet at the current execution state of the path. When the boolean value returned by the '*next()*' method is used in an if or while condition, this value states if the number of rows in the ResultSet is greater or equal to the number of times the '*next()*' method has been called so far on this ResultSet. In this case, *shelves.next()* will return true if the ResultSet *shelves* contains at least one row (as *shelves.next()* has been called once on the ResultSet):

```

; TAKING ELSE BRANCH OF: if ( ! shelves.next() )
(assert ( $\geq shelves1Size$  1))

```

Symbolic execution for update creates a new symbol (1) typed as an integer function, that will replace the previous symbol associating the attribute value to each object in

the table. As this new symbol is the second one to represent the value of the attribute *numberOfBooks*, it is named *numberOfBooks2*. A couple of constraints (2)(3) is then generated to relate the old and new attribute values in the table: one for the rows that do not match the WHERE condition (2), and one for those that do (3). Finally, constraints are added to specify that no integrity constraint was violated during the update. In this case, a constraint (4) is added to state that the updated attribute values still enforce the check constraint defined in the database schema.

```

; con.createStatement().execute(
; "UPDATE shelf SET numberOfBooks=numberOfBooks+1 WHERE id =" + theShelf);
(1) (declare-fun numberOfBooks2 (shelf) Int)
(2) (assert (forall ((p shelf))
      ( $\Rightarrow$  (or (and (shelf1 p) (not (= (id1 p) theshelf1))) (not (shelf1 p)))
        (= (numberOfBooks2 p) (numberOfBooks1 p))))))
(3) (assert (forall ((p shelf))
      ( $\Rightarrow$  (and (shelf1 p) (= (id1 p) theshelf1))
        (= (numberOfBooks2 p) (+ (numberOfBooks1 p) 1))))))
(4) (assert (forall ((a shelf))
      (> (numberOfBooks2 a) 0)))

```

Subsequently, as in our example the path specifies that the catch block of the try/catch statement must be executed, a constraint (1) is added to ensure that the program variables and the database are in a state where the INSERT execution will violate a schema integrity constraint. In this case, the constraint states that the inserted row has a similar primary key as the primary key of an existing row in the table or that the inserted row has a foreign key value that does not reference any existing row in the shelf table. Constraints are also automatically added to ensure that the ‘size()’ (2) and ‘get(int)’ (3) methods do not cause any runtime error.

```

; TAKING THE CATCH BRANCH OF:
; try { con.createStatement().execute(
; "INSERT INTO book VALUES (" + newBooks.get(i).intValue() + "," + theShelf + ")");
; } catch (SQLException e) {
(1) (assert (or (exists ((a book)) (and (book1 a)
      (= (code1 a) (select (elements newbooks1) i1))))
      (forall ((a shelf)) ( $\Rightarrow$  (shelf1 a)
        (not (= (id1 a) theshelf1))))))
(2) (assert (not (isNull newbooks1)))
(3) (assert ( $\geq$  i1 0))
(3) (assert (< i1 (size newbooks1)))

```

The content of the catch block is then symbolically executed:

```

; error = 1;
(declare-const error2 Int)
(assert (= error2 1))

```

As the path specifies that the else branch of the if statement must be executed this time, a constraint is generated to specify that the condition of the if should be false:

```

; TAKING ELSE BRANCH OF: if (error==0)
(assert (not (= error2 0)))

```

Symbolic execution for Rollback statements tells the algorithm to represent the current content of each database table using the symbols that were representing the content of the table just before the last start of a new transaction (saved by the algorithm at the beginning of the method execution and after each call to commit or abort). In this case, the database state is restored to its state at the method start, i.e. the algorithm rewinds the counters for the database symbols and symbols *book1*, *code1*, *shelfId1*, *shelf1*, *id1* and *numberOfBooks1* represent the content of the database after the ‘con.rollback()’ statement.

The assignment statement is then symbolically executed:

```
; i = i + 1;
(declare-const i2 Int)
(assert (= i2 (+ i1 1)))
```

As the path specifies that the loop body must not be executed any more, a constraint is generated to specify that, at this point in time, the loop condition should be false:

```
; ESCAPING LOOP: while ( !(newBooks == null) & (i < newBooks.size()) )
(assert (not (and (not (isNull newbooks1)) (< i2 (size newbooks1))))))
(assert (not (isNull newbooks1)))
```

As a return statement is met, the algorithm stops and returns the generated SMTv2 constraint model. The Z3 solver can now be asked to find a valuation for the defined symbols satisfying the constraints. As the algorithm records what symbols represent the initial, respectively final, values of a variable or table, the input and output values of the method (for the considered path) can easily be extracted from the solution to the constraint system.

For our example, the set of 31 constraints was solved by Z3 4.3.2 in 24ms (1.8 GHz Intel Core i5, 8GB Ram) and the test data that were obtained from the solution to the constraint system are summarized in the following tables:

Inputs			Outputs														
Name	Symbol(s)	Value	Name	Symbolic(s)	Value												
TABLE shelf	CONTENT: <i>shelf1</i> , ATTRIBUTES: <i>id1</i> <i>numberOfBooks1</i>	<table><tr><td>id</td><td>n.Books</td></tr><tr><td>6</td><td>1</td></tr><tr><td>12</td><td>1</td></tr></table>	id	n.Books	6	1	12	1	TABLE shelf	CONTENT: <i>shelf1</i> , ATTRIBUTES: <i>id1</i> <i>numberOfBooks1</i>	<table><tr><td>id</td><td>n.Books</td></tr><tr><td>6</td><td>1</td></tr><tr><td>12</td><td>1</td></tr></table>	id	n.Books	6	1	12	1
id	n.Books																
6	1																
12	1																
id	n.Books																
6	1																
12	1																
TABLE book	CONTENT: <i>book1</i> ATTRIBUTES: <i>code1</i> <i>shelfId1</i>	<table><tr><td>code</td><td>s.Id</td></tr><tr><td>4</td><td>12</td></tr></table>	code	s.Id	4	12	TABLE book	CONTENT: <i>book1</i> ATTRIBUTES: <i>code1</i> <i>shelfId1</i>	<table><tr><td>code</td><td>s.Id</td></tr><tr><td>4</td><td>12</td></tr></table>	code	s.Id	4	12				
code	s.Id																
4	12																
code	s.Id																
4	12																
newBooks	<i>newbooks1</i>	[4]	newBooks	<i>newbooks1</i>	[4]												
in.nextInt()	<i>theshelf1</i>	[6]	addedBooks	<i>addedbooks1</i>	[]												

7.3.2 Constraint Generation Rules

For sake of completeness, the following tables define the constraint generation rules used by our algorithm in case of an Insert (table 7.6), Update (table 7.7), Delete (table 7.8) and Add/Remove (table 7.9) statement. Table 7.10 explains the abbreviations used in the previous tables.

In each of these tables, any declaration of a new symbol leverages a generator providing a fresh symbol identifier, i.e. which has still not been used in the SMTv2 code generated so far. This is denoted by (for a function declaration):

(declare-fun freshSym Type)

and by (for a constant declaration):

(declare-const freshSym Type)

All the subsequent references to **freshSym** in the table represent this newly declared symbol. The generated fresh symbols are named according to the naming rule detailed along the

Table 7.6: SMT-Lib constraints generation rules for INSERT statements

INSERT INTO $\langle id \rangle$ VALUES ($\langle int-expr \rangle_1$, ... , $\langle int-expr \rangle_i$, ... , $\langle int-expr \rangle_n$)
<pre> if (<i>no exception thrown in path for this INSERT</i>) { ; <i>Inserted primary key value does not already exist</i> (assert(forall((a $\langle id \rangle$)) ($\Rightarrow$ (name($\langle id \rangle$) a) (not(= (name(pk) a) <i>smt2Of</i>($\langle int-expr \rangle_{pk^{pos}}$)))))) ; <i>Inserted values constrained by the i^{th} foreign key reference existing rows</i> (assert(exists((a fk_i^{tab})) (and(= (name(fk_i^{pk}) a) <i>smt2Of</i>($\langle int-expr \rangle_{fk_i^{pos}}$) (name($fk_i^{tab}$) a)))) ; <i>Symbol for new table content</i> (declare-fun <i>freshSym</i> ($\langle id \rangle$) Bool) ; <i>Constraints describing new table content</i> (assert (forall ((a $\langle id \rangle$)) ($\Rightarrow$ (name($\langle id \rangle$) a) (<i>freshSym</i> a)))) (assert (exists ((a $\langle id \rangle$)) (and (= (att_{i} a) <i>smt2Of</i>($\langle int-expr \rangle_i$) (<i>freshSym</i> a)))) (assert (forall ((a $\langle id \rangle$)) ($\Rightarrow$ (and(not (name($\langle id \rangle$) a)) (not (= (att_{i} a) <i>smt2Of</i>($\langle int-expr \rangle_i$))) (not (<i>freshSym</i> a)))))) ; <i>No duplicate inserted row</i> (assert (forall ((a $\langle id \rangle$) (b $\langle id \rangle$)) ($\Rightarrow$ (and(and (<i>freshSym</i> a) (<i>freshSym</i> b)) (= (pk a) (pk b)) (= a b)))) } else { // <i>Logical disjunction between every possible constraint</i> // <i>violation given the database schema and this insert:</i> ; <i>The inserted primary key value already exists in the table</i> (exists ((a $\langle id \rangle$)) (and (name($\langle id \rangle$) a) (= (name(pk) a) <i>smt2Of</i>($\langle int-expr \rangle_{pk^{pos}}$)))) ; <i>$i^{th}$ inserted foreign key value does not reference a row:</i> (forall ((a fk_i^{tab})) ($\Rightarrow$ (name(fk_i^{tab}) a) (not (= (name(fk_i^{pk}) a) <i>smt2Of</i>($\langle int-expr \rangle_{fk_i^{pos}}$)))) ; <i>An inserted attribute violates the i^{th} check constraint:</i> (not (<i>co_i^{right}</i> <i>smt2Of</i>($\langle int-expr \rangle_{co_i^{pos}}$))) } </pre>

example given in the previous subsection.

Finally, assertions are handled as if statements. For example :

```
assert x == 0;
```

is handled as:

```

if (x == 0)
  End of computation with AssertionError

```

Table 7.7: SMT-Lib constraints generation for UPDATE statements

UPDATE $\langle id \rangle$ SET $\langle id \rangle_{att} = \langle db-int-expr \rangle$ WHERE $\langle db-cond \rangle$
<pre> if (<i>no exception thrown in path for this UPDATE</i>) { ; <i>Symbol for new attribute values</i> (declare-fun freshSym ($\langle id \rangle$) Int) ; <i>Constraints describing new attribute values</i> (assert (forall ((a $\langle id \rangle$)) ($\Rightarrow$ (or (and (name($\langle id \rangle$) a) (not smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, a)) (not (name($\langle id \rangle$) a)) (= (freshSym a) (name($\langle id \rangle_{att}$) a)))))) (assert (forall ((a $\langle id \rangle$)) ($\Rightarrow$ (and (name($\langle id \rangle$) a) smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, a) (= (freshSym a) smt2Of($\langle db-int-expr \rangle$, $\langle id \rangle$, a)))))) ; <i>Update on attribute constrained by i^{th} foreign key does not let pending references</i> (assert (forall ((a $\langle id \rangle$)) ($\Rightarrow$ (name($\langle id \rangle$) a) (exists ((b fk$_i^{tab}$) (and (name(fk$_i^{tab}$) b) (= (freshSym a) (name(fk$_i^{pk}$) b))))))) ; <i>Update on attribute constrained by primary key does not let duplicate attribute values</i> (assert (forall ((a $\langle id \rangle$) (b $\langle id \rangle$)) ($\Rightarrow$ (and (and(name($\langle id \rangle$) a) (name($\langle id \rangle$) b)) (= (freshSym a) (freshSym b)) (= a b)))) ; <i>Update on primary key referenced by i^{th} foreign key does not let pending references</i> (assert (forall ((a ifk$_i^{tab}$) ($\Rightarrow$ (name(ifk$_i^{tab}$) a) (exists ((b $\langle id \rangle$)) (and (name($\langle id \rangle$) b) (= (name(ifk$_i^{att}$) a) (freshSym b))))))) ; <i>Update on attribute constrained by i^{th} check constraint does not violate the constraint</i> (assert (forall ((a $\langle id \rangle$)) (co$_i^{right}$ (freshSym a)))) } else { // <i>Logical disjunction between every possible constraint</i> // <i>violation given the database schema and this update:</i> ; <i>Update on primary key leads to duplicate attribute values</i> (exists ((a $\langle id \rangle$) (b $\langle id \rangle$)) (and(and (name($\langle id \rangle$) a) (and (name($\langle id \rangle$) b) (not (= a b)))) (or (and smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, a) (and smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, b) (= smt2Of($\langle db-int-expr \rangle$, $\langle id \rangle$, a) smt2Of($\langle db-int-expr \rangle$, $\langle id \rangle$, b)))) (and (not smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, a)) (and smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, b) (= (name($\langle id \rangle_{att}$) a) smt2Of($\langle db-int-expr \rangle$, $\langle id \rangle$, b)))))) ; <i>Update on primary key referenced by the i^{th} foreign key lets pending references</i> (exists ((a $\langle id \rangle$) (b $\langle id \rangle$)) (and (and (name($\langle id \rangle$) a) (name(ifk$_i^{tab}$) b)) (and (and (not (= (name($\langle id \rangle_{att}$) a) smt2Of($\langle db-int-expr \rangle$, $\langle id \rangle$, a)) (= (name($\langle id \rangle_{att}$) a) (name(ifk$_i^{pk}$) b)) smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, a)))))) ; <i>Update on attribute constrained by i^{th} foreign key lets pending references</i> (exists ((a $\langle id \rangle$)) (and (and (name($\langle id \rangle$) a) smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, a) (not (exists ((b name(fk$_i^{tab}$))) (= (name(fk$_i^{pk}$) b) smt2Of($\langle db-int-expr \rangle$, $\langle id \rangle$, a)))))) ; <i>Update on attribute constrained by i^{th} check constraint violates the constraint</i> (exists ((a $\langle id \rangle$)) (and (and (name($\langle id \rangle$) a) smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, a) (not (co$_i^{right}$ smt2Of($\langle db-int-expr \rangle$, $\langle id \rangle$, a)))))) } </pre>

Table 7.8: SMT-Lib constraints generation for DELETE statements

DELETE FROM $\langle id \rangle$ WHERE $\langle db-cond \rangle$
<pre> if (<i>no exception thrown in path for this DELETE</i>) { ; <i>Symbol for new table content</i> (declare-fun freshSym ($\langle id \rangle$) Bool) ; <i>Constraints describing new table content</i> (assert (forall ((a $\langle id \rangle$)) (= (freshSym a) (and (name($\langle id \rangle$) a) (not smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, a)))))) ; <i>Delete does not let pending references for i^{th} foreign key</i> (assert(forall ((a fk_i^{tab}) (b $\langle id \rangle$)) ($\Rightarrow$ (and (name($\langle id \rangle$) b) (and (not (freshSym b)) (name(ifk$_i^{tab}$) a))) (not (= (name(pk) b) (name(ifk$_i^{att}$) a)))))) } else { // <i>Logical disjunction between every possible constraint</i> // <i>violation given the database schema and this update:</i> ; <i>Delete lets pending references for i^{th} foreign key</i> (exists ((a fk_i^{tab}) (b $\langle id \rangle$)) (and (and (and (name($\langle id \rangle$) b) (name(ifk$_i^{tab}$) a)) smt2Of($\langle db-cond \rangle$, $\langle id \rangle$, b)) (= (name(pk) b) (name(ifk$_i^{att}$) a)))) } </pre>

Table 7.9: SMT-Lib constraints generation rules for add(int) and remove(int) statements

$\langle id \rangle$.add($\langle int-expr \rangle$);
<pre> (declare-const freshSym BoundedList) (assert (not (isNull name($\langle id \rangle$)))) (assert (not (isNull freshSym))) (assert (= (size freshSym) (+ (size name($\langle id \rangle$)) 1))) (assert (= (elements freshSym) (store (elements name($\langle id \rangle$)) (size name($\langle id \rangle$)) smt2Of($\langle int-expr \rangle$)))) </pre>
$\langle id \rangle$.remove($\langle int-expr \rangle$);
<pre> (declare-const freshSym BoundedList) (assert (not (isNull name($\langle id \rangle$)))) (assert (not (isNull freshSym))) (assert ($\geq$ (size name($\langle id \rangle$)) 1)) (assert (= (size freshSym) (- (size "+ oldVar+") 1))) (assert ($\geq$ smt2Of($\langle int-expr \rangle$) 0)) (assert ($<$ smt2Of($\langle int-expr \rangle$) (size name($\langle id \rangle$)))) (assert (forall ((i Int)) ($\Rightarrow$ (and ($\geq$ i 0) ($<$ i smt2Of($\langle int-expr \rangle$))) (= (select (elements name($\langle id \rangle$)) i) (select (elements freshSym) i)))))) (assert (forall ((i Int)) ($\Rightarrow$ (and ($\geq$ i smt2Of($\langle int-expr \rangle$)) ($<$ i (size freshSym))) (= (select (elements name($\langle id \rangle$)) (+ i 1)) (select (elements freshSym) i)))))) </pre>

Table 7.10: SMT-Lib Translation Abbreviations List

Abbreviation	Meaning
$smt2Of(x)$	Java condition/expression x translated into a corresponding SMTv2 condition/expression.
$smt2Of(x, t, r)$	SQL condition/expression x evaluated for row r in table t translated into a corresponding SMTv2 condition/expression.
$name(x)$	if (x refers to a database table name) then The symbol that represents the current content of table x else if (x refers to a database attribute name) The symbol that represents the current values of attribute x else if (x refers to a Java variable name) The symbol that represents the current content of the Java variable x
att_i	Name of the i_{th} attribute in the list of attributes of table $\langle id \rangle$
pk	Name of the primary key attribute of table $\langle id \rangle$.
pk^{pos}	Position of primary key in the list of attributes of table $\langle id \rangle$
fk_i^{tab}	Name of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id \rangle$
fk_i^{pk}	Name of the primary key attribute of the table referenced by the i_{th} foreign key in the list of foreign keys of table $\langle id \rangle$
fk_i^{pos}	Position of the foreign key attribute, declared by the i_{th} foreign key in the list of table $\langle id \rangle$, in the list of attributes of table $\langle id \rangle$
ifk_i^{tab}	Name of the table where is declared the i_{th} foreign key referencing table $\langle id \rangle$ in the whole schema
ifk_i^{att}	Name of the foreign key attribute declared by the i_{th} foreign key referencing table $\langle id \rangle$ in the schema
co_i^{pos}	Position of the attribute constrained by the i_{th} check constraint declared in table $\langle id \rangle$
co_i^{right}	Inverted right part of the i_{th} check constraint declared in table $\langle id \rangle$ (i.e. inverted right part of “ $a > 0$ ” is “ < 0 ”)

Chapter 8

Experimental Evaluation

Contents

8.1	Experimental Framework	151
8.1.1	A Tool for Relational Testing	151
8.1.2	Sample Code for Experimentation	151
8.2	Evaluating the Alloy and Z3 Solvers for Relational Testing .	152
8.2.1	Performance over Commonly Crafted Samples	152
8.2.2	Practical Scalability Limits for Z3	154
8.3	Benchmarking Relational Testing against Normalization . . .	158
8.4	Additional Experiments over Open-Source Code	158

IN this chapter, we detail how our symbolic execution algorithm has been used within a test generation tool to provide an experimental answer to the raised research questions about relational testing.

Section 8.1 details the integration of our relational symbolic execution algorithm into a test generation tool and discusses the sample code on which this tool was experimented. Section 8.2 describes the experimental results obtained while evaluating the Alloy and Z3 solvers for relational testing. Section 8.3 details a performance comparison between relational testing and the normalization approach for testing SQL applications. Section 8.4 presents some additional evaluation of relational testing over open-source code.

8.1 Experimental Framework

8.1.1 A Tool for Relational Testing

In order to evaluate experimentally our relational testing approach, we have built a test input generation tool for SimpleDB programs, based on the relational symbolic execution algorithm detailed in the previous chapter. By using this tool to generate test data for a set of selected SQL applications, we were then able to perform the necessary measurements and comparisons to provide an experimental answer to the main research questions about relational testing.

Practically, the tool integrates the proposed relational symbolic execution algorithm with a static path exploration process. Given a SimpleDB program to test, it performs a depth-first traversal of the control-flow graph in a static way and generates either Alloy or SMT-Lib constraints in parallel, using the adequate version of the relational symbolic execution algorithm. These constraints are then solved either by the Alloy or Z3 solver, to find test inputs for the explored paths. For each feasible path, input (and output) data (including database content) are extracted from the obtained constraint solution and recorded as constituting a test case.

Once all the paths have been explored, the test set comprising all the generated test cases is returned to the user. The traversal depth was bounded to paths that execute the body of each loop in the program at most K times, where K is a parameter of the tool. Consequently, the test set generated by our tool satisfy a finitely applicable variant of the common path-coverage criterion [Zhu et al., 1997], similar to the loop count- K criterion originally proposed in [Howden, 1975].

From a technical standpoint, the tool was coded in Java 1.6 and run on a dual core Intel Core i5 processor at 1.8GHz (256 KB L2 cache per core and 3 MB L3 cache) with 8GB of dual channel DDR3 memory at 1600 MHz. The runtime environment was the Oracle JVM 1.6.0_45 under a 32-bit edition of Microsoft Windows 8.1. The version 4.3.0 of the Microsoft Z3 solver was used. In order to offer a fair performance comparison with related work, some experiments were repeated on an Intel Pentium 4 processor at 3.06GHz with 1GB of RAM, under a 32-bit edition of Microsoft Windows XP.

8.1.2 Sample Code for Experimentation

The sample code used for our experimental evaluation of relational testing is composed of eighteen Java methods, performing SQL operations over relational databases. These methods can be divided into four groups:

- The first group contains nine SimpleDB programs. Each program was crafted to systematically evaluate the correct symbolic execution of one of the different Java and SQL constructs available in SimpleDB. As such, the methods in this group exercise the different behaviors of the integer and list operators, conditional and loop statements, SQL statements and transaction management primitives.
- The second group contains three SimpleDB programs crafted to implement realistic use cases offered by OLTP software. The first method in this group performs repeated manipulations of integers and lists to compare and save some data. The second method performs many interleaved reads and writes in a database containing four

tables, representing regular or prospect customers that make purchases of products. The third method mixes SQL statements with traditional Java code and uses SQL transactions. The manipulated database contains two tables that represent authors writing theater plays.

- The third and fourth groups contain Java methods extracted respectively from UnixUsage and RiskIt, the two open-source software that have been used in [Pan et al., 2013b], as a basis for evaluating their test generation approach based on SQL normalization. As a reminder, UnixUsage is a monitoring application for Unix, manipulating a database with eight tables and thirty one attributes. RiskIt is an insurance rate estimation application, manipulating a database with thirteen tables and fifty-seven attributes.

Together, the eighteen methods from our testbed constitute a set of five hundreds lines of code, containing notably eighty SQL statements (including SELECT, INSERT, UPDATE, DELETE statements, as well as transaction management code), over databases containing up to thirteen tables (subject to primary key, foreign key and check constraints).

Detailed statistics for each of these methods can be found in table 8.1, including the value of K used in our tool to limit the loop exploration depth for each method. Given this value of K , the number of paths to be explored is provided for each method, detailing the number of feasible and unfeasible ones. The high number of infeasible paths is principally due to the methods in the second group (and, to a lesser extend, the methods for testing conditional constructs from the first group), methods that were particularly constructed in this way for testing the soundness of our test generation tool.

Finally, the code of each of the eighteen methods can be found in appendix B.

8.2 Evaluating the Alloy and Z3 Solvers for Relational Testing

8.2.1 Performance over Commonly Crafted Samples

Our first experimental objective was to evaluate the ability of the Alloy and Z3 solvers to solve effectively and efficiently the constraints produced by relational testing. In order to do so, we used the Alloy and SMT-Lib versions of our tool to generate test cases for the methods of the first and second groups from our testbed.

The Alloy solver basically solves the constraints by setting upper bounds on the scope of the possible integers, as well as on the cardinality of the possible different rows which can appear in the solution for each table. This makes finite the set of possible solutions, which can then be exhaustively explored, by transforming the constraints into an equivalent SAT instance, solved by a dedicated SAT solver. As a consequence, if no solution is found with the default minimal bound value, our tool should repeat the process recursively on a higher default bound value, until a time-out is reached. When the time-out is reached, the tool considers the underlying path as unfeasible. The time-out value should thus be long enough to enable a proper model finding for feasible paths. In the same time, it should not be too long to detect infeasible paths in minimal time. Finding an optimal time-out value appeared to be very difficult in practice, as it depends on the size and complexity of the constraints to solve.

Table 8.1: Statistics for the selected samples

Name	LOCs	Number of SQL statements	Number of tables	K Depth	Number of Paths	
					Feasible	Infeasible
<i>SIMPLEDB SAMPLES</i>						
- SimpleDB Language Unit Testing						
Integers	7	0	0	3	1	0
Lists	9	0	0	3	1	0
If and While	30	0	0	1	1	319
Conditions	30	0	0	1	1	319
Select	23	7	2	3	1	63
Insert	14	7	2	3	1	7
Update	20	11	2	3	1	15
Delete	6	3	2	3	1	1
Transactions	13	2	2	3	2	6
- Realistic OLTP Methods						
Handling Data	45	1	1	5	23	3513
Clients and Products	56	18	4	5	4	32
Play Catalog	72	4	2	2	63	5569
<i>SAMPLES EXTRACTED FROM OPEN-SOURCE SOFTWARE</i>						
- UnixUsage						
courseNameExists	7	1	8	1	2	0
getCommandsByCategory	10	1	8	1	2	0
getCourseIDByName	10	1	8		2	0
getDepartmentIDByName	11	1	8	1	2	0
- RiskIt						
createNewUser	91	7	13	2	3	21
deleteUsers	55	16	13	2	2	181
ALL METHODS	509	80	up to 13	up to 5	113	10046

On the left part of table 8.2, we detail the results obtained with the Alloy version of the tool, over the feasible paths of the methods from groups one and two. For each method, the table provides the minimal bound values making it possible to find inputs for all the feasible paths in the method. It also provides the total number of constraints generated for these feasible paths and the minimal time within which the solver was able to solve these constraints. The methods named “Integers”, “Update”, “Delete” and “Clients and Products” involve either large integer values or repeated actions on a single table. As a consequence, finding inputs requires large enough bound values for the scope of possible different integer values or for the cardinality of different row values from the particular table. Bound values enabling one to generate inputs for these methods in reasonable time were in practice found manually, using a costly trial and error approach. Moreover, the results show that once the bounds are increased, the size of the SAT instance generated by the Alloy solver quickly explodes, while the solving time blows up from dozens of milliseconds to several minutes. Finding inputs for the “Handling Data” and the “Play Catalog” methods, with their thousands of paths, revealed to be intractable in reasonable time, with the Alloy solver and without any manual help.

In accordance with [El Ghazi and Taghdiri, 2011], where Z3 is used as a complement to the Alloy solver for proving instances unsatisfiable, the Z3 solver was able to detect almost instantaneously the unsatisfiability of the generated constraints. As a consequence, the SMT-Lib version of the tool was able to detect properly and efficiently the thousands of unfeasible paths in the methods from groups one and two. As we used a more recent version of Z3 than in [El Ghazi and Taghdiri, 2011], involving notably the new model-based quantifier instantiation technique for solving quantified constraints, we could hope that the solver would also be able to find efficiently satisfying models for the generated constraints and thus produce inputs for satisfiable paths. The obtained results are detailed on the right part of table 8.2, both for feasible and infeasible paths. The Z3 solver was able to find inputs for each feasible path in milliseconds, always outperforming the Alloy solver, even when minimal bound values were used in the Alloy version. As an example, the Alloy version took more than thirty eight minutes to find test inputs for the four feasible paths of method “Clients and Products”, when the Z3 version was able to do it in three seconds and a half. Finding inputs for the “Handling Data” and the “Play Catalog” methods, which was intractable using Alloy, became possible in less than two minutes using Z3.

8.2.2 Practical Scalability Limits for Z3

Our second objective with the experiments was to estimate the practical scalability limits of current solver technology for the generated constraints. In order to do so, we have measured the number of generated constraints and the constraint solving time for relational symbolic execution (using Z3) of the following method, containing a single execution path, traversing a linear block of SQL statements. These statements have been selected to provide a balanced mix of typical queries and DML statements. DML statements act on fields subject to integrity constraints.

```
CREATE TABLE t1 (  
  idt1 INTEGER NOT NULL,  
  fieldt1 INTEGER NOT NULL,  
  CONSTRAINT t1PK PRIMARY KEY (idt1),  
  CHECK(idt1 > 0));
```

```

CREATE TABLE t2 (
idt2 INTEGER NOT NULL,
fieldt2 INTEGER NOT NULL,
CONSTRAINT t2PK PRIMARY KEY (idt2),
CONSTRAINT t2FK FOREIGN KEY (fieldt2) REFERENCES t1(idt1),
CHECK(idt2 > 0));

```

```

void test (Connection con,Scanner in) throws SQLException {
int i = 1;
con.createStatement().execute("INSERT INTO t1 VALUES (" + i + ", " + i + ")");
con.createStatement().execute("INSERT INTO t1 VALUES (" + (i + 1) + ", " + (i + 1) + ")");
con.createStatement().execute("INSERT INTO t2 VALUES (" + i + ", " + i + ")");
con.createStatement().execute("INSERT INTO t2 VALUES (" + (i + 1) + ", " + (i + 1) + ")");
int input1 = in.nextInt();
ResultSet result1 = con.createStatement().executeQuery("SELECT idt1
FROM t1
WHERE fieldt1=" + i);
result1.next();
con.createStatement().execute("DELETE FROM t2 WHERE idt2=" + input1);
con.createStatement().execute("UPDATE t2
SET fieldt2 = 1 + fieldt2
WHERE idt2 < " + (2 + result1.getInt("idt1")));
input1 = in.nextInt();
con.createStatement().execute("DELETE FROM t1 WHERE idt1=" + input1); }

```

Then, the measurement was repeated for a similar path, but where the block of SQL statements was executed twice in turn: a first time normally and a second time with a value of i increased by 2. The statements of the second round were executed directly on the database resulting from the first round so that they can modify the rows it contains as well. The process was then repeated with four rounds and so on.

The obtained measurements are reported in Figures 8.1 and 8.2. These graphs show the number of constraints and the constraint solving time, as a function of the number of SQL statements executed in the path. The number of constraints evolves linearly with the number of SQL statements. The constraint solving time increases exponentially with the number of SQL statements. These results are coherent with the complexity of our constraint generation rules and of SMT solving algorithms. They provide an estimate value of the practical scalability of the approach, where the solving time starts to rise around 60 statements and becomes important above 100 statements, on our Intel Core i5 at 1.8GHz.

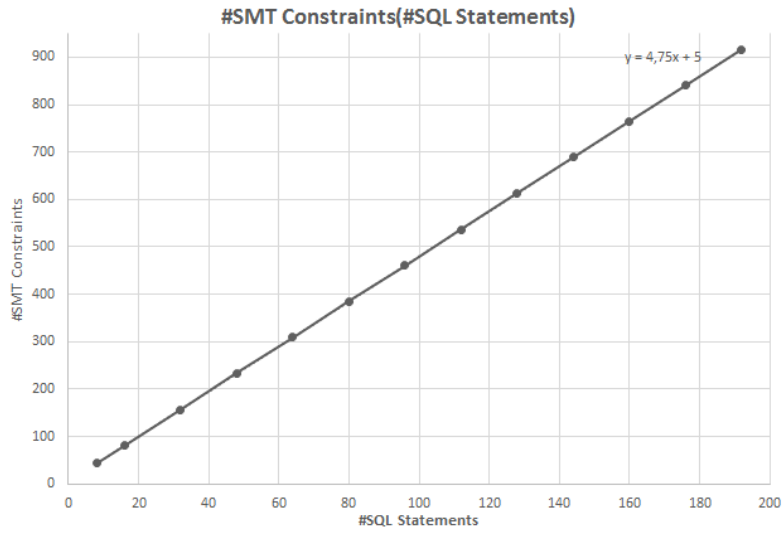


Figure 8.1: Constraint number as a function of the number of SQL statements in the path

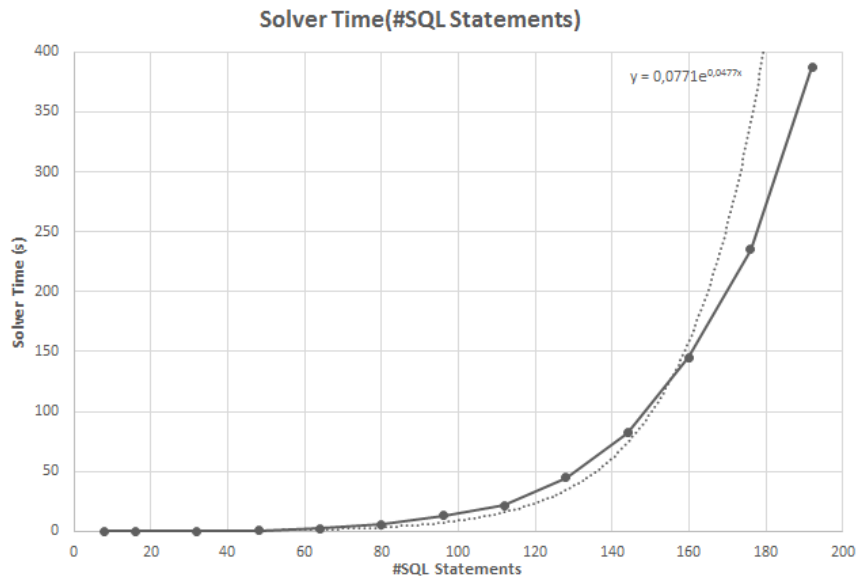


Figure 8.2: Solving time as a function of the number of SQL statements in the path

Table 8.2: Statistics for the selected samples

Name	Feasible Paths to Alloy			All Paths to SMT-Lib			
	Cardinality Bound	Cons- -traits	Solver Time	Constraints	Solver Time	Unfeasible Paths	Solver Time
<i>SIMPLEDB SAMPLES</i>							
- SimpleDB Language Unit Testing							
Integers	1 but 5 int	6	129ms	4	16ms	0	0ms
Lists	4	10	186ms	45	16ms	0	0ms
If and While	1	17	46ms	15	15ms	4241	3s 989ms
Conditions	1	19	37ms	19	16ms	5325	4s 345ms
Select	3	41	221ms	61	47ms	4131	2s 355ms
Insert	1	23	40ms	22	16ms	214	124ms
Update	7 but 8 int	39	6m 6s 299ms	41	63ms	703	765ms
Delete	1 but 8 int	16	51s 638ms	15	15ms	17	16ms
Transactions	1	38	137ms	44	30ms	148	93ms
- Realistic OLTP Methods							
Handling Data	-	-	-	2154	641ms	487120	1m 3s 504ms
Clients and Products	8 but 16 clients	502	38m 37s 506ms	612	3s 470ms	4068	7s 587ms
Play Catalog	-	-	-	4343	1s 361ms	527625	1m 35s 67ms

8.3 Benchmarking Relational Testing against Normalization

Our last experimentation objective was to compare the performance of relational testing with the SQL normalization approach proposed in [Pan et al., 2013b]. As relational testing enables generating tests without increasing the number of paths to explore in the code, a performance bump could be expected. In order to achieve this goal, we have selected four simple Java methods from the testbed used by [Pan et al., 2013b] and used our relational testing tool on them. These methods are those of the third group from our testbed.

Our tool was asked to generate tests with the same statement coverage as offered by [Pan et al., 2013b]. In order to offer a fair comparison, the total generation time for each method was measured on a Pentium 4 configuration, similar to the one used in [Pan et al., 2013b]. The obtained results are detailed in the first part of table 8.3. Results for the normalization-based tool are extracted from [Pan et al., 2013b]. Our tool based on relational testing was one order of magnitude faster than the tool based on normalization, generating tests in a few seconds compared to more than one minute and a half for the other tool.

It should be noted that [Pan et al., 2013b] is built upon the Microsoft Pex tool for constraint-based testing, which makes use of a dynamic path exploration and provides some support for character strings. Our research prototype tool makes use of a static path exploration and string attributes appearing in the tested database were transformed into integer ones (which is not a significant problem as the tested methods use no string-specific operator). These differences between the two tools may have an impact over our performance comparison between the relational testing and normalization approach. However, we think that this impact must be limited in this case, as the tested methods are very small, involving between 7 and 11 simple statements, with only one simple SQL (SELECT) statement, one loop and no conditional per method. As a consequence, we think that the detailed differences cannot explain by themselves the measured performance gap.

8.4 Additional Experiments over Open-Source Code

UnixUsage and RiskIt provide a useful source of open-source code for evaluating our relational testing tool. As a consequence, we have also extracted some methods involving not only SELECT statements, as in [Pan et al., 2013b], but also SQL statements writing into the database, in order to assess our tool. These methods are those of the fourth group from our testbed.

Test data generation for these methods required to extend our symbolic execution algorithm to handle some new parts of Java and SQL, used in RiskIt, or to simulate them in SimpleDB. In particular, the management of tables with no primary key or with multiple-attribute primary key was integrated in the algorithm and string management operations were simulated using either integers or lists of integers. As the second part of table 8.3 shows, correct test data were generated in just a few seconds and, moreover, the generated test data enabled detecting a possible defect in the code of RiskIt, where a runtime error is thrown when the method `createNewUser` is called on inputs where the inserted job does not reference any existing occupation or industry.

Table 8.3: Statistics for the selected samples

Name	Analysis via SynDB [Pan et al., 2013b]		Analysis via Relational Symbolic Execution	
	Covered Statements	Analysis Time	Covered Statements	Analysis Time
SAMPLES EXTRACTED FROM OPEN-SOURCE SOFTWARE				
- <i>UnixUsage</i>				
courseNameExists	7	1m 41s	7	4s 984ms
getCommandsByCategory	10	1m 30s	10	1s 454ms
hline getCourseIDByName	10	1m 33s	10	2s 891ms
getDepartmentIDByName	11	1m 43s	11	1s 578ms
- <i>RiskIt</i>				
createNewUser	-	-	91	11s 294ms
deleteUsers	-	-	55	5s 39ms

Chapter 9

Discussion

Contents

9.1	Synthesis of Research Contributions	163
9.2	Comparison with Related Work	163
9.2.1	SQL Normalization into Native Code	163
9.2.2	Query Results as Input Arrays	164
9.2.3	Qex	164
9.3	Threats to Validity	165
9.3.1	Internal Validity	165
9.3.2	External Validity	165

IN this chapter, we discuss and evaluate our research contributions, based on the obtained experimental results and a proper comparison with related work. Section 9.1 synthesizes the research contributions made in this thesis. Section 9.2 compares our relational testing approach and tool with the three directly related approaches and tools. Section 9.3 discusses the internal and external validity of our research contributions.

9.1 Synthesis of Research Contributions

In this work, we have introduced relational testing, a new approach for generalizing constraint-based testing to SQL applications. Relational testing is original as it returns to the relational and logical roots of SQL, to enable a direct symbolic execution of SQL code into constraints. A relational symbolic execution was developed, handling a core Java/SQL sublanguage for OLTP use cases, which we have defined and called SimpleDB. Both the Alloy and Z3 versions of this algorithm were integrated into a prototype test generation tool, based on a static path exploration. This prototype was used to perform experiments over eighteen SQL applications, including open-source code.

Our experiments have shown that Alloy-based relational testing was impractical, due to the exhaustive search, within a parametric bounded scope, performed by the Alloy solver. On the contrary, Z3-based relational testing has been shown able to generate test cases for all the considered applications, seamlessly and in reasonable time. These results provide both an experimental confirmation and new elements to the research presented in [El Ghazi and Taghdiri, 2011], where Z3 was used to prove the unsatisfiability of Alloy constraints. Our experiments can indeed be seen as some kind of case study for [El Ghazi and Taghdiri, 2011], which confirm their conclusions for proving unsatisfiable instances. Moreover, our results show that versions of Z3 more recent than the one used in [El Ghazi and Taghdiri, 2011] include new model-based quantifier instantiation techniques, able to outperform the Alloy solver also in model finding for satisfiable instances.

Our work demonstrates that relational testing of SQL applications implementing OLTP use-cases, composed of a limited number of SQL statements acting on a limited number of rows, is possible with current solver technologies. However, our scalability measurements show that the approach may face exponential solving time blow-up, if pieces of code involving a large number of SQL statements had to be tested, with current technologies.

9.2 Comparison with Related Work

9.2.1 SQL Normalization into Native Code

Conceptually, normalizing SQL code into native code and then applying classical constraint-based testing on the result is an alternate approach to ours, where the SQL code is compiled into relational constraints during symbolic execution. Replacing a single SQL statement by a piece of native code, simulating its execution by a DBMS, may dramatically increase the number of paths to be explored, compared to the original code [Chan and Cheung, 1999]. In contrast, in relational testing, the execution paths to be considered are limited to paths in the *original* program code, promising a potential performance bump.

Our experimental benchmarks, comparing the tool from [Pan et al., 2013b], based on normalization, with our relational testing tool, tend to confirm the existence of such a performance bump. Moreover, even if the symbolic execution of INSERT, UPDATE and DELETE statement is conceptually possible using normalization, [Pan et al., 2013b] only validates their approach over code containing SELECT statements. Relational testing is thus the only approach experimented in presence of SQL code writing into the database, as well as in presence of SQL transactions.

On the other hand, the tool proposed in [Pan et al., 2013b] provides important features that are not handled by ours. First, it deals with more complex SELECT queries. Secondly, it relies on Microsoft Pex, as an off-the-shelf back-end constraint-based testing tool. First,

Pex handles natively a large part of the C# language, where our tool is restricted to a small part of Java. Secondly, Pex provides some support for character strings, which constitute an important datatype in SQL applications, not handled by our tool. Thirdly, Pex uses a dynamic path exploration process with heuristics for exploring the control-flow graph, making concretization possible if necessary, and thus notably giving a direct access to the SQL code crafted dynamically by the application. As a consequence, the tool from [Pan et al., 2013b] can account, at least to some extent, for dynamically crafted SQL statements, where our tool only considers SQL statements whose structure is completely defined statically.

9.2.2 Query Results as Input Arrays

The approach proposed in [Emmi et al., 2007], which transforms query results into input arrays, exhibits some conceptual limitations compared to relational testing. It indeed provides no conceptual handling for INSERT, UPDATE and DELETE statements, neither for transaction management primitives, which are crucial components of database applications. Moreover, [Emmi et al., 2007] only provides a conceptual intuition and no proper mechanism for taking into account the data integrity constraints defined in the database schema. [Pan et al., 2013b] advocates that such a lack leads to the possible generation of invalid test data and to a poor code coverage. Finally, it is unclear how [Emmi et al., 2007] will handle code containing successive queries, whose result is not independent from each other (like the simple example of code containing twice the same query), as the approach seems to consider each query result as a totally independent input.

Nevertheless, the tool detailed in [Emmi et al., 2007] has important features that are not present in ours. First, and contrary to our tool, it deals with null values and, at least partly, with character strings. Secondly, it relies on the JCute constraint-based testing tool. JCute handles a large part of the Java language, including multi-threading, and uses a dynamic path exploration process. As with Pex, this makes the tool from [Emmi et al., 2007] able to benefit from concretization as well as to account, at least to some extent, for dynamically crafted SQL statements.

9.2.3 Qex

A substantial amount of work has been done on how to generate test database content exhibiting some desirable properties, given only the database schema and a set of queries to be executed over the database. The main difference between our work and these approaches is that they essentially work without considering the control flow of the applications manipulating the database.

Qex [Veanes et al., 2009, Veanes et al., 2010] is probably the one of these techniques which is the closest to our work, as both approaches are based on the translation of the SQL semantics into SMT-Lib constraints, solved using Z3. However, the two approaches exhibit also important differences. Firstly, Qex only considers input generation for a single SQL SELECT query in isolation, but, on the other hand, the handled queries can be more complex than in our tool. Secondly, the two techniques translate the SQL semantic into constraints differently: our tool use predicates and quantifiers to represent relations, while Qex use fixed-size lists of tuples. Qex involves thus a mechanism similar to the one used in the Alloy solver, where the solving process must be repeated on increasing size for the input relations, in order to find a solution. Such an approach does not enable proving

the constraints unsatisfiable. It should be noted that Qex was developed at a time where model-based quantifier instantiation techniques, aimed at finding satisfiable models for quantified constraints, were still not available in Z3.

9.3 Threats to Validity

9.3.1 Internal Validity

Our performance comparison between relational testing and SQL normalization into native code revealed a very strong advantage for relational testing, over similar hardware configurations. However, the comparison was performed on a limited number of small samples, using different test computers and considering tools with different architectures. We think that the impact of the two last elements is minor, given the small size of the tested methods, and cannot explain, on its own, how important the measured performance gap is. However, these elements, as well as the lack of a comparison over a large number of various complex pieces of code, threaten our ability to conclude, from our experimental work, that relational testing actually improves the testing time, compared to SQL normalization.

9.3.2 External Validity

Our relational testing tool has been designed in the framework of SimpleDB programs. As a consequence, it handles only a limited subset of the Java/SQL syntax and considers applications implementing OLTP use cases, i.e. small pieces of code involving a limited number of SQL statements, acting on a limited number of rows. Several elements threaten the generalizability of our approach to handle any application containing SQL code.

Conceptually, relational testing can be integrated into any existing general-purpose constraint-based testing tool, based on the principle of symbolic execution, to provide this tool with the ability to handle SQL code. Such an integration into existing state-of-art tools, based on advanced techniques to handle a wide range of applications written in a traditional programming language, would strongly improve the practical scope of relational testing. However, such an integration has not been experimented. Moreover, despite the important amount of research behind these tools, their ability to scale over very large applications seems not guaranteed with current technology.

The handling of a larger SQL language may require the generation of constraints which could be hard to manage by the solver, like non-linear arithmetic constraints. Moreover, as SQL is a complex language, fragmented into several dialects, developing a universal SQL symbolic execution engine, able to handle any piece of SQL code, will be a very hard task. Our tool provides no direct way to handle SQL code dynamically crafted by the application. Alternate tools have shown that the use of a dynamic path exploration process makes it possible to alleviate the problem, thanks to the availability of the concrete value of the built SQL code. However, such an approach fails to handle cases where (parts of) the SQL code are inputs of the tested application.

Finally, current solver technologies have been measured to face a scalability issue when required to solve the constraints generated by the symbolic execution of a large number of SQL statements. Whether this problem can be solved or at least alleviated, by optimizing the constraint generation or by new solving capabilities, remains an open research question.

Part III

A Path for the Future of Relational Testing

“It’s a dangerous business, Frodo, going out your door. You step onto the road, and if you don’t keep your feet, there’s no knowing where you might be swept off to.”

BILBO BAGGINS in *The Lord of The Rings* by J.R.R. Tolkien

Chapter 10

From SimpleDB towards Full-Scale SQL Applications

Contents

10.1 Sample Code for Discussion	171
10.2 Integration with Constraint-Based Testing Tools	171
10.3 Handling Dynamically-Crafted SQL	173
10.4 Building and Solving Constraints for SQL	173
10.5 Detecting SQL Runtime Errors	174
10.6 Handling Non-Deterministic SQL	175

THE previous chapter has identified the main threats to the generalizability of our relational testing approach, from the SimpleDB language to full-scale SQL applications. In this chapter, we identify and discuss five main research directions sketching a path towards relational testing of a wider scope of SQL applications. We think that following this path could make an evaluation of the approach in an industrial context possible. Elements of the discussion presented here have been published in [Marcozzi et al., 2014a].

Section 10.1 presents a sample SQL application, used as a motivating example in the remainder of the chapter. Section 10.2 discusses the integration of relational testing within existing constraint-based testing tools. Section 10.3 examines the handling of dynamically-crafted SQL code. Section 10.4 considers constraint generation and solving for bigger pieces of SQL code, written using a wider range of the SQL syntax. Section 10.5 treats the question of SQL runtime errors, which can occur frequently in SQL applications. Section 10.6 glances at non-deterministic behaviors in the SQL language.

10.1 Sample Code for Discussion

In this introductory section, we present the sample faulty SQL application from figure 10.1 as a basis for the subsequent discussion over the future of relational testing. This application describes code as it could have been extracted from bookshop software. It is composed of the Java method *removeBookShelf* and of the SQL DDL code describing the part of the bookshop database touched by the method, i.e. the table *shelf*. In this table, a shelf is described by a technical identifier (an integer), by its label (a string of maximum 50 characters), by the number of books it contains (an integer which must be positive) and by the moment where it is planned to be reordered (which can be *null*). When a book is definitely removed from the bookshop, several Java methods are called in the bookshop's software to update the bookshop's database. Among those, the *removeBookShelf* method is supposed to decrease by one the number of books in the shelf from which the book was removed. If this book was the last one, the shelf is removed from the database. In practice, the *removeBookShelf* method receives the label of the shelf and a connection to the database and retrieves the current number of books in the shelf from the database. If this number is greater than one, it is updated in the database, otherwise the shelf's row is deleted. Three SQL-related defects are present in this application. First, the method mistakenly supposes that the SELECT query will always return one and only one row. Secondly, the UPDATE statement decreases the number of books by two instead of one. Finally, there is a syntax error in the DELETE statement (doubled WHERE token).

10.2 Integration with Constraint-Based Testing Tools

The Java-handling part of the relational testing tool presented in this thesis is very basic, compared to the current state of the art. First, it handles a very limited part of Java, where a tool like Pex encompasses a large part of the object-oriented language C#. Secondly, it is based on a static path exploration process, where a dynamic path exploration process is more powerful, as it provides a direct access for concretization. Thirdly, it uses a basic exhaustive depth-first search of the control-flow graph, where current tools use several techniques, like heuristics, to handle path explosion.

However, our tool was only a prototype, demonstrating how a classical symbolic execution mechanism for a typical programming language could be empowered with the ability to generate constraints in presence of SQL code. As this integration provided promising results, it should now be repeated with a state-of-art tool based on symbolic execution. The resulting tool would keep its advanced features, like a large syntax coverage for a modern programming language, as well as a dynamic and explosion-proof path exploration process, but would gain the ability to handle SQL statements in the tested code. Moreover, as discussed in the next section, coupling relational testing with a dynamic path exploration process would make easier to handle dynamically-crafted SQL code.

Microsoft Pex is a particularly appropriate candidate for integration with our work, as it is based as well on the Z3 solver and benefits from Microsoft support. In addition, such an integration could enable a deeper comparison with the SQL normalization-based tool from [Pan et al., 2013b], which is also based on Pex. The Pex tool is not open-source but provides an extension interface and some academic teams have published papers extending Pex. At our best knowledge, available open-source constraint-based testing tools are CREST and KLEE (for C code) as well as Symbolic Path Finder (for Java bytecode).

Figure 10.1: A faulty SQL application that removes a book from a shelf in a bookshop database.

```
CREATE TABLE shelf (
  id INTEGER NOT NULL,
  label NVARCHAR2(50) NOT NULL,
  numberOfBooks INTEGER NOT NULL,
  nextReordering TIMESTAMP(2),
  CONSTRAINT sPK
    PRIMARY KEY (id),
  CHECK (numberOfBooks > 0));
```

```
1 void removeBookShelf (Connection con,int theShelf) throws SQLException {
2   String condition = " WHERE label =" + theShelf;
3   ResultSet rs = con.createStatement().executeQuery("SELECT numberOfBooks FROM shelf" + condition);
4   rs.next();
5   int booksNumber = rs.getInt("numberOfBooks");
6   if (booksNumber > 1) {
7     con.createStatement().execute("UPDATE shelf SET numberOfBooks=numberOfBooks-2" + condition);
8   } else {
9     con.createStatement().execute("DELETE FROM shelf WHERE" + condition);
10  }
11 }
```

10.3 Handling Dynamically-Crafted SQL

The symbolic execution algorithm used in our tool was designed in the context of *static SQL*, whereas the *removeBookShelf* method from Figure 10.1 uses *dynamic SQL*, making the direct use of the constraint generation rules from chapter 7 impossible for testing this application. In static SQL, the syntactic structure of the SQL statement is completely defined statically. The value of the constants in the typed expressions used in this statement can be defined parametrically at compile time as a function of the application’s variables. In dynamic SQL, the application builds dynamically a character string containing the SQL statement’s code, which is parsed and executed at runtime. Conceptually, symbolic execution is only possible for static SQL, as the symbolic execution engine will always need the syntactic structure of the SQL code, as well as a precise definition of the relation enforced by the SQL code between the application’s variables and the database’s content, in order to be able to generate constraints. Nevertheless, in practice, the dynamic SQL code can often be normalized, for a given execution path in the code, into an equivalent static SQL code. This normalization is even easier if a possible runtime value for the string variable containing the SQL code is known for the considered path, thanks to the use of a dynamic path exploration process. As a simple illustrating example, if we consider a path in the *removeBookShelf* method that we would like to execute symbolically, the dynamic *SELECT* statement can be replaced by a static SQL statement “SELECT numberOfBooks FROM shelf WHERE **label** = X” where the value of parameter *X* is defined as equal to the value of the *theShelf* parameter. The constraint generation rules from chapter 7 can then be applied on this normalized version of the path’s code.

However, path normalization of dynamic SQL into static SQL is not always possible, as dynamic SQL allows to use the application’s inputs as parts of the syntactic structure of the dynamically-crafted SQL code. In some applications, the whole SQL statement can even be loaded as a string from the database itself. Such cases are particularly problematic, as symbolic execution computes inputs from code analysis and is thus not designed for applications whose code is part of the input. Integration between symbolic execution and partial evaluation [Jones et al., 1993] could be an interesting research direction for solving this problem. Partial evaluation basically optimizes a piece of code by precomputing statically all the parts of the code that depend on inputs which are known at compile time. In an SQL application, by choosing appropriate concrete values for those parts of the inputs that are used to define the syntactic structure of the dynamic SQL statements, one could use partial evaluation to produce representative specialized versions of the original application that can be properly evaluated symbolically. Interleaving symbolic execution and partial evaluation has already been studied in another context by [Bubel et al., 2010]. Detecting which parts of the application’s inputs should be made concrete could benefit from existing work (e.g. [Halfond et al., 2006, Su and Wassermann, 2006]) over detection of SQL injections attacks.

10.4 Building and Solving Constraints for SQL

Our experiments have shown that our proposed technique faces scalability issues if the constraints generated from large numbers of SQL statements have to be solved. In such cases and even if solver performance improvements can be expected in the future, symbolic execution of SQL should be tailored to generate constraints expressed in a form which has been optimized to be handled very efficiently by the internal algorithms used in the solver.

Building such optimal constraint generation rules is made difficult by the fact that the syntax and semantics of SQL is large and complex, and can vary strongly in practice between different DBMS's versions and manufacturers. A research direction for overcoming these difficulties would be to use relational algebra as an intermediate language for constraint generation. The symbolic execution engine would compile the original SQL code into a minimal relational algebra, and then the algebraic operators would be translated into logical constraints, using a minimal set of translation rules optimized for the solver. Algorithms translating SQL statements into equivalent combinations of a core set of relational algebra's operators have already been developed, in the context of DBMS design [Connolly and Begg, 2005]. In practice, this idea should be refined, as SQL is more powerful than relational algebra, since it allows non-relational constructs like rows ordering and aggregation, null values etc. The intermediate language should thus be extended by a minimal set of operators for describing common and tractable non-relational parts of SQL. *Concretization* could be the last-ditch solution to handle exotic or too complex parts of SQL.

SQL applications, like the one of figure 10.1, can mix various operations over various datatypes, such as strings, binary objects, numeric values and timestamps. Conversions between these types are also common, like at line 2 of the *removeBookShelf* method from Figure 10.1, where an integer is converted into a string. Symbolic execution of these operations requires to benefit from solvers able to handle quantified SMT constraints combining, in potentially new ways, hard parts of well-studied Σ -theories, like the non-linear integer arithmetic, or string-length constraints, with new Σ -theories like a timestamp Σ -theory.

Hopefully, research over constraint solving is a particularly dynamic domain, nourished by practical applications, like relational testing is. Research is ongoing (e.g. [Shannon et al., 2009, Bjørner et al., 2009, Saxena et al., 2010, Redelinghuys et al., 2012, Tateishi et al., 2013, Zheng et al., 2013, Liang et al., 2014]) towards a proper solving of string constraints inter-related with other kinds of constraints, in the context of symbolic execution. Similarly, recent works (see e.g. [Jovanovic and de Moura, 2012]) have considered solving non-linear integer arithmetic constraints. Other works have also studied the particular problem of multi-granularity temporal constraint solving (see e.g. [Bettini et al., 2002]).

10.5 Detecting SQL Runtime Errors

Runtime errors can be frequent in SQL applications, as there is no compile-time integration between the application's code and the DBMS's code. When the DBMS encounters an error while processing an application's request, it warns the application by throwing a runtime error. Some of these errors can be caused by a defect in the application, like submitting a syntactically wrong dynamic SQL code, misusing the DBMS's API, accessing an empty or closed result set, violating the database's integrity constraints or writing values in a different format than specified by the database's schema. Relational testing should thus treat every SQL statement (and DBMS's API call) as a kind of switch/ case statement dealing with a set of possible runtime errors, typically of the form:

```
switch (SQL Statement) {
case violates the  $i^{th}$  integrity constraint of the schema :
  throw new SQLException("Constraint ...violated");
case contains a syntax error :
  throw new SQLException("Syntax error: ...");
```

...

default:

Execute statement. }

Two major challenges are, first, to detect, for a given statement, what kinds of runtime errors can occur, and, secondly, to infer the constraints that drive the execution towards a particular kind of runtime error being raised or not. The symbolic execution algorithm proposed in this thesis already contains a partial implementation of this mechanism, limited to handling simple INSERT/UPDATE/DELETE statements violating primary key, foreign key or check constraints.

As an example, symbolic execution should detect that the *getInt* method at line 6 of the *removeBookShelf* method from Figure 10.1 can throw an *SQLException*, if called on an empty *ResultSet*. If we model each datatype in the application by integers and replace the dynamically-crafted SELECT statement by equivalent static SQL, the generation rules from chapter 7 can be applied to produce constraints driving the execution towards such a faulty *ResultSet* access. Solving these constraints would show that they are satisfiable (for example, if the database is empty at method's start, whatever the other inputs are) and enables thus one to detect the fault concerning the *SELECT* statement, which can return less than one row for some inputs. Similarly, satisfiable constraints can be generated for the path leading to a violation of the database's CHECK constraint by the UPDATE statement at line 8 (violation occurs if the selected shelf contains two books), which is a hint at the fault signaled for the UPDATE statement (with a correct code, decreasing the number of books by one, no constraint violation would be possible and the generated constraints would be unsatisfiable). Finally, considering the integration of relational testing with a dynamic path exploration process, any concrete execution of the method taking the *else* branch at line 11 will throw an *SQLException*, because of the syntax error in the DELETE statement. In order to be able to build inputs triggering such SQL syntax errors, we could notably get inspired by [Gould et al., 2004], which studies statical detection of syntax errors in dynamic SQL code.

10.6 Handling Non-Deterministic SQL

Non-determinism creeps in many places in the SQL semantics. A common example is the undefined order in which a SELECT statement can return the selected rows. For example, let us consider two runs of the *removeBookShelf* method from Figure 10.1 over 66 as input value for parameter *theShelf* and the following content for the table *shelf*:

id	label	numberOfBooks	nextReordering
1	"66"	560	9/9/15 11:00:00
3	"66"	1	8/9/15 9:10:00

Both rows are selected by the SELECT query at line 3 but let us suppose that they are returned in different order between the two runs. During the first run, the value 560 is returned by the *getInt* method at line 6, triggering the execution of the *then* branch of the *if* statement. During the second run, the value 1 is returned by the *getInt* method, triggering the execution of the *else* branch of the *if* statement.

Non-deterministic SQL is problematic for constraint-based testing, as it enables a single test case to trigger different execution paths in the code and possibly to produce different outputs. Such a non-deterministic behavior of the whole application should be detected as

it can be a hint at an underlying design defect in the code. In our example, the defect hinted at is the previously signaled fault concerning the `SELECT` statement, which can return more than one row for some inputs.

A common way to handle such a non-determinism in constraint-based testing is to constrain the order in which the rows are returned by the `SELECT` statement as if it was one of the method's inputs. In such a way, any solution to a path-constraint would define a particular row order and thus trigger the execution of a single path in the code, producing a single possible output. Moreover, one can subsequently execute the method by varying the row order, while keeping the other inputs constant, to detect if the taken path and the produced output are affected.

Other well-known examples of non-deterministic SQL are queries based on the current date in temporal databases, whose result can vary depending on when the query is executed, or insertion of new rows using fields values randomly selected by the DBMS. In both cases, the current date and the randomly selected field values can be considered as inputs of the tested application as well.

Chapter 11

Possible Extensions and New Challenges

THIS last chapter briefly relates, in a prospective way, some possible research extensions and new research challenges, around the relational testing approach introduced in this work.

Using Dedicated Code Coverage Criteria

Several works have proposed code coverage criteria tailored towards a proper testing of SQL applications. Our approach being somehow parameterized with respect to the paths that should be considered, one can hope that integrating it with these criteria should be possible with little effort. Such an integration could help driving the path exploration process towards interesting parts of the control-flow graph, in applications exhibiting a large number of paths.

Taking Advantage of DDL and Business Use Cases

Relational testing makes use of structural information about the tested application (its control-flow graph) to generate test cases. The underlying principle is that a test set is adequate if it covers a sufficient part of the application's structure. Using complementary sources of information to drive the test generation could be a fruitful extension of our approach. For example, we could try to generate test cases covering the structure of the database DDL schema or of the business use cases specifying the application.

Extracting Test Data from an Existing Database

Our test generation approach for SQL applications generates from scratch a synthetical input content for the database. However, many SQL applications are developed to interact with an existing database, which is already in production. Testing such SQL applications with real inputs, extracted from their future host database, would made the testing process more relevant. In the same time, the test data should still cover as much as possible of the different behaviors (i.e. execution paths) of the tested application, in order to detect defects properly.

Optimization-driven constraint solving could be a possible way to satisfy these two requirements. As a constraint system typically admits many different satisfying models, optimization-driven constraint solving deals with finding a satisfying model which optimizes

a given objective function. In this case, such an approach could be used to find models for the constraints generated by relational testing, modulo an objective function that favors models using some typical data from the host database, extracted thanks to a statistical analysis of its content.

Search-based testing could be another way to select covering inputs from an existing database. Search-based testing is an alternate test generation approach that searches the application's input set for covering data, using search heuristics. A search-based testing approach for SQL applications could use data from the existing database as seeds for its search heuristics.

Minimizing Database Modifications between Tests

Supposing a dynamic path exploration process, testing an SQL application requires to run the code over some input content for the database, then solve the obtained constraints to find a new input content for the database, and finally repeat the process recursively over this new content. Between the end of a code run and the beginning of the subsequent one, the database content at the end of the first run must thus be replaced by the one obtained by solving the constraints. Such an operation can be costly in time with a large database and a large number of records. Optimization-driven constraint solving could be used to find a satisfying model for the constraints, so that the modifications to apply to the database are minimal, given its content at the end of the first run.

Considering other Data Technologies

In this work, we have considered a JDBC-based interaction with a relational database. Even if these technologies are popular, testing applications based on some other interaction protocols and database models (like NoSQL or ORM-based) is a promising new challenge.

Conclusion

CODE defects can cause important software failures, provoking potentially serious damage for the society. In current software engineering practice, testing remains the primary approach to detect such defects, but is also a complex and costly activity, so that its automation is an interesting mean to improve software cost and reliability. In recent years, constraint-based testing has emerged as a promising automated approach to generate representative sets of inputs for testing applications. This approach has notably been used in an industrial context at NASA and Microsoft. In the same time, barely few works have studied how to generate automatically test inputs, including database states, to evaluate the correct SQL-based interaction between an application and a relational database. As more and more data are collected about all aspects of personal, social and economical life, such research, providing an effective mean to locate sources of data corruptions, would avoid increasingly damaging consequences for the society.

In this work, we have introduced relational testing as a new approach for providing the current state of the art in constraint-based testing with the ability to handle SQL code in the tested application. The main originality of this approach was to return to the relational and logical roots of SQL, to enable a direct symbolic execution of SQL code into constraints. Such a direct symbolic execution of SQL makes it possible to test applications both reading and writing in the database, and without modifying the tested code. As this is not generally possible using alternate approaches, relational testing could benefit either from a wider applicability or a better performance than these approaches. However, it was unclear whether relational testing could be implemented in practice, as it was unknown how the current relational constraint solvers (Alloy and Z3) could handle effectively and efficiently the constraints produced for SQL code in combination with normal code.

In order to evaluate the practical applicability of relational testing and compare its performance with alternate techniques, we have used a clear methodological approach. First, we have carefully selected the limits of our study. In order to do so, we have defined SimpleDB, a subset of Java/SQL to write SQL applications, and we have restrained the study to SimpleDB programs implementing transactional business use cases. Then, we have built a symbolic execution algorithm for SimpleDB code, based on the principle of relational testing. Both the Alloy and Z3 versions of this algorithm were integrated into a prototype test generation tool. This prototype was finally used to perform the necessary experiments over eighteen SQL applications, including open-source code.

These experiments revealed that Alloy-based relational testing was impractical, but that Z3-based relational testing could generate test cases for all the considered applications, seamlessly and in reasonable time. However, it appeared that the approach could face scalability problems for pieces of code involving larger number of SQL statements. Finally and as expected, a performance bump was measured in favor of our tool, compared to the relevant alternate approach, on similar hardware configuration.

The obtained results enable us to conclude that relational testing is a promising approach,

as it was able to generate tests quickly on simple but already realistic applications, and as it strongly outperformed the relevant alternate approach. However, our relational testing tool also misses important features compared to the alternate ones and has still a limited scope in practice. More fundamentally, several challenges still need to be overcome to be able to apply the approach in an industrial context. In the same time, a deeper comparison should be realized with the relevant alternate approach, to confirm the performance bump that we have measured, over a more various panel of applications.

In the future, we think that automating testing of most SQL applications should be possible with reasonable effort. In order to do so, we have identified five main research directions to follow. First, we should integrate the principle of relational testing in a general purpose, state-of-art, constraint-based testing tool. Secondly, we should handle dynamically-crafted SQL using concretization, like in alternate approaches, and partial evaluation. Thirdly, we should generate tractable constraints for a larger part of SQL, using an intermediate language and tailoring the constraint generation for the internal solver algorithms. Improvements in solver performance, driven by relational testing needs, could also be hoped. Fourthly, we should detect properly SQL runtime errors, using efficient error analysis, for translating them into constraints. Finally, we should deal with non-deterministic SQL by making the non-deterministic choice an application input. We are convinced that these directions sketch a path towards the future of relational testing, as an entirely realistic and useful approach for testing SQL applications.

Bibliography

- [Abiteboul et al., 1995] Abiteboul, S., Hull, R. and Vianu, V. (1995). Foundations of Databases. Addison-Wesley. 8, 18, 41, 111
- [Abran et al., 2001] Abran, A., Bourque, P., Dupuis, R. and Moore, J. W., eds (2001). Guide to the Software Engineering Body of Knowledge - SWEBOK 3.0. IEEE Press, Piscataway, NJ, USA. 83
- [Althaus et al., 2009] Althaus, E., Kruglov, E. and Weidenbach, C. (2009). Superposition Modulo Linear Arithmetic SUP(LA). In Frontiers of Combining Systems, (Ghilardi, S. and Sebastiani, R., eds), vol. 5749, of Lecture Notes in Computer Science pp. 84–99. Springer Berlin Heidelberg. 73
- [Appelt et al., 2014] Appelt, D., Nguyen, C. D., Briand, L. C. and Alshahwan, N. (2014). Automated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach. In Proceedings of the 2014 International Symposium on Software Testing and Analysis ISSTA 2014 pp. 259–269, ACM, New York, NY, USA. 105
- [Arkoudas et al., 2004] Arkoudas, K., Khurshid, S., Marinov, D. and Rinard, M. (2004). Integrating Model Checking and Theorem Proving for Relational Reasoning. In Relational and Kleene-Algebraic Methods in Computer Science, (Berghammer, R., Möller, B. and Struth, G., eds), vol. 3051, of Lecture Notes in Computer Science pp. 21–33. Springer Berlin Heidelberg. 79
- [Armando and Giunchiglia, 1993] Armando, A. and Giunchiglia, E. (1993). Embedding complex decision procedures inside an interactive theorem prover. Annals of Mathematics and Artificial Intelligence 8, 475–502. 60
- [Artzi et al., 2008] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A. and Ernst, M. D. (2008). Finding Bugs in Dynamic Web Applications. In Proceedings of the 2008 International Symposium on Software Testing and Analysis ISSTA '08 pp. 261–272, ACM, New York, NY, USA. 93
- [Baars et al., 2011] Baars, A. I., Harman, M., Hassoun, Y., Lakhota, K., McMinn, P., Tonella, P. and Vos, T. E. J. (2011). Symbolic search-based testing. In ASE, (Alexander, P., Pasareanu, C. S. and Hosking, J. G., eds), pp. 53–62, IEEE. 95
- [Bachmair et al., 2003] Bachmair, L., Tiwari, A. and Vigneron, L. (2003). Abstract Congruence Closure. Journal of Automated Reasoning 31, 129–168. 60, 61
- [Barrett et al., 2011] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A. and Tinelli, C. (2011). CVC4. In Proceedings of the 23rd International Conference on Computer Aided Verification CAV'11 pp. 171–177, Springer-Verlag, Berlin, Heidelberg. 59, 70

- [Barrett et al., 2013] Barrett, C., Deters, M., de Moura, L. M., Oliveras, A. and Stump, A. (2013). 6 Years of SMT-COMP. *J. Autom. Reasoning* 50, 243–277. 57
- [Barrett et al., 2009] Barrett, C., Sebastiani, R., Seshia, S. and Tinelli, C. (2009). Satisfiability Modulo Theories. In *Handbook of Satisfiability*, (Biere, A., Heule, M. J. H., van Maaren, H. and Walsh, T., eds), vol. 185, chapter 26, pp. 825–885. IOS Press. 55, 56, 60, 61, 63, 64, 65, 73, 205
- [Barrett et al., 2010] Barrett, C., Stump, A. and Tinelli, C. (2010). The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, (Gupta, A. and Kroening, D., eds), SMT-Lib. 57
- [Barrett and Tinelli, 2007] Barrett, C. and Tinelli, C. (2007). CVC3. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, (Damm, W. and Hermanns, H., eds), vol. 4590, of *Lecture Notes in Computer Science* pp. 298–302, Springer-Verlag, Berlin, Germany. 92
- [Berre and Parrain, 2010] Berre, D. L. and Parrain, A. (2010). The Sat4j library, release 2.2. *JSAT* 7, 59–6. 47, 78
- [Bettini et al., 2002] Bettini, C., Wang, X. and Jajodia, S. (2002). Solving multi-granularity temporal constraint networks. *Artificial Intelligence* 140, 107 – 152. 174
- [Biere, 2008] Biere, A. (2008). PicoSAT Essentials. *JSAT* 4, 75–97. 47
- [Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E. M. and Zhu, Y. (1999). Symbolic Model Checking Without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems TACAS '99* pp. 193–207, Springer-Verlag, London, UK, UK. 48
- [Binnig et al., 2006] Binnig, C., Kossmann, D. and Lo, E. (2006). Testing Database Applications. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data SIGMOD '06* pp. 739–741, ACM, New York, NY, USA. 103
- [Binnig et al., 2007] Binnig, C., Kossmann, D. and Lo, E. (2007). Reverse Query Processing. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on* pp. 506–515, IEEE Computer Society Press. 103
- [Binnig et al., 2008] Binnig, C., Kossmann, D. and Lo, E. (2008). Multi-RQP: Generating Test Databases for the Functional Testing of OLTP Applications. In *Proceedings of the 1st International Workshop on Testing Database Systems DBTest '08* pp. 5:1–5:6, ACM, New York, NY, USA. 103, 119
- [Binnig et al., 2007] Binnig, C., Kossmann, D., Lo, E. and Özsu, M. T. (2007). QAGen: generating query-aware test databases. In *SIGMOD Conference*, (Chan, C. Y., Ooi, B. C. and Zhou, A., eds), pp. 341–352, ACM. 104
- [Björner and Pichora, 1998] Björner, N. and Pichora, M. (1998). Deciding fixed and non-fixed size bit-vectors. In *Tools and Algorithms for the Construction and Analysis of Systems*, (Steffen, B., ed.), vol. 1384, of *Lecture Notes in Computer Science* pp. 376–392. Springer Berlin Heidelberg. 61

- [Bjørner et al., 2009] Bjørner, N., Tillmann, N. and Voronkov, A. (2009). Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems* pp. 307–321. Springer. 94, 174
- [Bonacina et al., 2009] Bonacina, M. P., Lynch, C. and Moura, L. (2009). On Deciding Satisfiability by $DPLL(\Gamma + \mathcal{T})$ and Unsound Theorem Proving. In *Proceedings of the 22Nd International Conference on Automated Deduction CADE-22* pp. 35–50, Springer-Verlag, Berlin, Heidelberg. 69, 70, 73
- [Boonstoppel et al., 2008] Boonstoppel, P., Cadar, C. and Engler, D. (2008). RWset: Attacking Path Explosion in Constraint-based Test Generation. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS’08/ETAPS’08* pp. 351–366, Springer-Verlag, Berlin, Heidelberg. 93
- [Boyer and Moore, 1990] Boyer, R. and Moore, J. (1990). A theorem prover for a computational logic. In *10th International Conference on Automated Deduction*, (Stickel, M., ed.), vol. 449, of *Lecture Notes in Computer Science* pp. 1–15. Springer Berlin Heidelberg. 60
- [Bradley et al., 2006] Bradley, A. R., Manna, Z. and Sipma, H. B. (2006). What’s Decidable About Arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation VMCAI’06* pp. 427–442, Springer-Verlag, Berlin, Heidelberg. 61
- [Braunstein et al., 2005] Braunstein, A., Mézard, M. and Zecchina, R. (2005). Survey Propagation: An Algorithm for Satisfiability. *Random Struct. Algorithms* 27, 201–226. 54
- [Bryant, 1986] Bryant, R. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *Computers, IEEE Transactions on* C-35, 677–691. 53
- [Bryant et al., 1999] Bryant, R., German, S. and Velez, M. (1999). Exploiting Positive Equality in a Logic of Equality with Uninterpreted Functions. In *Computer Aided Verification*, (Halbwachs, N. and Peled, D., eds), vol. 1633, of *Lecture Notes in Computer Science* pp. 470–482. Springer Berlin Heidelberg. 60
- [Bryant et al., 2009] Bryant, R. E., Kroening, D., Ouaknine, J., Seshia, S. A., Strichman, O. and Brady, B. (2009). An abstraction-based decision procedure for bit-vector arithmetic. *International journal on software tools for technology transfer* 11, 95–104. 78
- [Bubel et al., 2010] Bubel, R., Hähnle, R. and Ji, R. (2010). Interleaving Symbolic Execution and Partial Evaluation. In *Proceedings of the 8th International Conference on Formal Methods for Components and Objects FMCO’09* pp. 125–146, Springer-Verlag, Berlin, Heidelberg. 173
- [Büning and Lettmann, 1999] Büning, H. and Lettmann, T. (1999). Propositional Logic: Deduction and Algorithms. *Cambridge tracts in theoretical computer science*, Cambridge University Press. 47, 49

- [Burnim and Sen, 2008] Burnim, J. and Sen, K. (2008). Heuristics for Scalable Dynamic Test Generation. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering ASE '08 pp. 443–446, IEEE Computer Society, Washington, DC, USA. 92
- [Cadaru et al., 2008] Cadaru, C., Dunbar, D. and Engler, D. (2008). KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation OSDI'08 pp. 209–224, USENIX Association, Berkeley, CA, USA. 92
- [Cadaru et al., 2006] Cadaru, C., Ganesh, V., Pawlowski, P. M., Dill, D. L. and Engler, D. R. (2006). EXE: Automatically generating inputs of death. In In Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS ACM. 88, 89, 92, 93
- [Cadaru et al., 2011] Cadaru, C., Godefroid, P., Khurshid, S., Pasareanu, C. S., Sen, K., Tillmann, N. and Visser, W. (2011). Symbolic Execution for Software Testing in Practice: Preliminary Assessment. In Proceedings of the 33rd International Conference on Software Engineering ICSE '11 pp. 1066–1071, ACM, New York, NY, USA. 60, 85, 92, 94
- [Cadaru and Sen, 2013] Cadaru, C. and Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communication of the ACM* 56, 82–90. 85, 89, 93, 94
- [Chan and Cheung, 1999] Chan, M. Y. and Cheung, S. C. (1999). Testing database applications with SQL semantics. In In Proceedings of the 2nd International Symposium on Cooperative Database Systems for Advanced Applications pp. 363–374, Springer. 99, 163
- [Chays et al., 2004] Chays, D., Deng, Y., Frankl, P. G., Dan, S., Vokolos, F. I. and Weyuker, E. J. (2004). An AGENDA for testing relational database applications: Research Articles. *Softw. Test. Verif. Reliab.* 14, 17–44. 104
- [Chays et al., 2008] Chays, D., Shahid, J. and Frankl, P. G. (2008). Query-based test generation for database applications. In Proceedings of the 1st international workshop on Testing database systems DBTest '08 pp. 6:1–6:6, ACM, New York, NY, USA. 104
- [choco Team, 2010] choco Team (2010). choco: an Open Source Java Constraint Programming Library. Research report 10-02-INFO École des Mines de Nantes. 92
- [Church, 1936] Church, A. (1936). An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics* 58, 345–363. 56
- [Codd, 1970] Codd, E. F. (1970). A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 377–387. 9, 18
- [Codd, 1971] Codd, E. F. (1971). Further Normalization of the Data Base Relational Model. IBM Research Report, San Jose, California *RJ909*. 9
- [Cohen et al., 2009] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W. and Tobies, S. (2009). VCC: A Practical System for Verifying Concurrent C. In Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics TPHOLs '09 pp. 23–42, Springer-Verlag, Berlin, Heidelberg. 60

- [Collingbourne et al., 2011] Collingbourne, P., Cadar, C. and Kelly, P. H. (2011). Symbolic Crosschecking of Floating-point and SIMD Code. In Proceedings of the Sixth Conference on Computer Systems EuroSys '11 pp. 315–328, ACM, New York, NY, USA. 93
- [Connolly and Begg, 2005] Connolly, T. and Begg, C. (2005). Database Systems: A Practical Approach to Design, Implementation, and Management. Number vol. 1 in International computer science series, Addison-Wesley. 8, 14, 25, 36, 37, 42, 174
- [Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In Proceedings of the third annual ACM symposium on Theory of computing pp. 151–158, ACM. 48
- [Cooper, 1972] Cooper, D. C. (1972). Theorem proving in arithmetic without multiplication. *Machine Intelligence* 7, 91–99. 56, 67
- [CVC4, 2014] CVC4 (2014). CVC4 String: <http://cvc4.cs.nyu.edu/wiki/Strings>. Website. 59
- [Da Cunha and Greathead, 2007] Da Cunha, A. D. and Greathead, D. (2007). Does personality matter?: an analysis of code-review ability. *Communications of the ACM* 50, 109–112. 84
- [Darwen and Date, 1995] Darwen, H. and Date, C. J. (1995). The Third Manifesto. *SIGMOD Rec.* 24, 39–49. 8
- [Davis et al., 1962] Davis, M., Logemann, G. and Loveland, D. (1962). A Machine Program for Theorem-proving. *Commun. ACM* 5, 394–397. 48
- [Davis and Putnam, 1960] Davis, M. and Putnam, H. (1960). A Computing Procedure for Quantification Theory. *J. ACM* 7, 201–215. 48
- [de la Riva et al., 2010] de la Riva, C., Suárez-Cabal, M. J. and Tuya, J. (2010). Constraint-based test database generation for SQL queries. In Proceedings of the 5th Workshop on Automation of Software Test AST '10 pp. 67–74, ACM, New York, NY, USA. 104
- [de Moura and Bjørner, 2008] de Moura, L. and Bjørner, N. (2008). Model-based Theory Combination. *Electron. Notes Theor. Comput. Sci.* 198, 37–49. 60, 205
- [De Moura and Bjørner, 2008] De Moura, L. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'08/ETAPS'08 pp. 337–340, Springer-Verlag, Berlin, Heidelberg. 70, 80, 93, 115
- [De Moura and Bjørner, 2011] De Moura, L. and Bjørner, N. (2011). Satisfiability Modulo Theories: Introduction and Applications. *Commun. ACM* 54, 69–77. 57, 59, 60, 64, 65, 72
- [Deng et al., 2005] Deng, Y., Frankl, P. and Chays, D. (2005). Testing Database Transactions with AGENDA. In Proceedings of the 27th International Conference on Software Engineering ICSE '05 pp. 78–87, ACM, New York, NY, USA. 104

- [Detlefs et al., 2005] Detlefs, D., Nelson, G. and Saxe, J. B. (2005). Simplify: A Theorem Prover for Program Checking. *J. ACM* 52, 365–473. 68
- [Dowson, 1997] Dowson, M. (1997). The Ariane 5 Software Failure. *SIGSOFT Softw. Eng. Notes* 22, 84–. 82
- [D’silva et al., 2008] D’silva, V., Kroening, D. and Weissenbacher, G. (2008). A Survey of Automated Techniques for Formal Software Verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 27, 1165–1178. 84
- [Duran and Ntafos, 1984] Duran, J. W. and Ntafos, S. (1984). An Evaluation of Random Testing. *Software Engineering, IEEE Transactions on* SE-10, 438–444. 95
- [Dutertre and de Moura, 2006a] Dutertre, B. and de Moura, L. (2006a). A Fast Linear-arithmetic Solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification CAV’06* pp. 81–94, Springer-Verlag, Berlin, Heidelberg. 61
- [Dutertre and de Moura, 2006b] Dutertre, B. and de Moura, L. (2006b). A Fast Linear-arithmetic Solver for DPLL(T). In *Proceedings of the 18th International Conference on Computer Aided Verification CAV’06* pp. 81–94, Springer-Verlag, Berlin, Heidelberg. 92
- [Eén and Biere, 2005] Eén, N. and Biere, A. (2005). Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Theory and Applications of Satisfiability Testing, (Bacchus, F. and Walsh, T., eds), vol. 3569, of Lecture Notes in Computer Science* pp. 61–75. Springer Berlin Heidelberg. 52
- [Eén and Sörensson, 2003] Eén, N. and Sörensson, N. (2003). An Extensible SAT-solver. In *6th International Conference, SAT 2003 Selected Revised Papers* pp. 502–518, Springer Berlin Heidelberg. 47, 78
- [El Ghazi and Taghdiri, 2010] El Ghazi, A. A. and Taghdiri, M. (2010). Analyzing alloy constraints using an SMT solver: a case study. In *5th International Workshop on Automated Formal Methods (AFM)*. Edinburgh ACM. 79
- [El Ghazi and Taghdiri, 2011] El Ghazi, A. A. and Taghdiri, M. (2011). Relational reasoning via SMT solving. In *Proceedings of the 17th international conference on Formal methods FM’11* pp. 133–148, Springer-Verlag, Berlin, Heidelberg. 59, 74, 79, 115, 138, 141, 154, 163
- [Emmi et al., 2007] Emmi, M., Majumdar, R. and Sen, K. (2007). Dynamic test input generation for database applications. In *Proceedings of the 2007 international symposium on Software testing and analysis ISSTA ’07* pp. 151–162, ACM, New York, NY, USA. 100, 101, 105, 164
- [Floratou et al., 2012] Floratou, A., Teletia, N., DeWitt, D. J., Patel, J. M. and Zhang, D. (2012). Can the Elephants Handle the NoSQL Onslaught? *Proc. VLDB Endow.* 5, 1712–1723. 43
- [Fraser and Zeller, 2010] Fraser, G. and Zeller, A. (2010). Mutation-driven Generation of Unit Tests and Oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis ISSTA ’10* pp. 147–158, ACM, New York, NY, USA. 95

- [Frias et al., 2007] Frias, M. F., Pombo, C. G. L. and Moscato, M. M. (2007). Alloy Analyzer+PVS in the Analysis and Verification of Alloy Specifications. In Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'07 pp. 587–601, Springer-Verlag, Berlin, Heidelberg. 79
- [Ganesh and Dill, 2007] Ganesh, V. and Dill, D. L. (2007). A Decision Procedure for Bit-vectors and Arrays. In Proceedings of the 19th International Conference on Computer Aided Verification CAV'07 pp. 519–531, Springer-Verlag, Berlin, Heidelberg. 92
- [Garcia-Molina et al., 2008] Garcia-Molina, H., Ullman, J. D. and Widom, J. (2008). Database Systems: The Complete Book. 2 edition, Prentice Hall Press, Upper Saddle River, NJ, USA. 9
- [Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA. 48
- [Ge and Moura, 2009] Ge, Y. and Moura, L. (2009). Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories. In Proceedings of the 21st International Conference on Computer Aided Verification CAV '09 pp. 306–320, Springer-Verlag, Berlin, Heidelberg. 70, 71
- [Ghilardi et al., 2005] Ghilardi, S., Nicolini, E. and Zucchelli, D. (2005). A Comprehensive Framework for Combined Decision Procedures. In Frontiers of Combining Systems, (Gramlich, B., ed.), vol. 3717, of Lecture Notes in Computer Science pp. 1–30. Springer Berlin Heidelberg. 60
- [Godefroid, 2011] Godefroid, P. (2011). Higher-order Test Generation. SIGPLAN Not. 46, 258–269. 90, 92
- [Godefroid et al., 2005] Godefroid, P., Klarlund, N. and Sen, K. (2005). DART: directed automated random testing. SIGPLAN Not. 40, 213–223. 89, 92
- [Godefroid et al., 2012] Godefroid, P., Levin, M. Y. and Molnar, D. (2012). SAGE: Whitebox Fuzzing for Security Testing. Queue 10, 20:20–20:27. 93
- [Gödel, 1931] Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. Monatshefte für mathematik und physik 38, 173–198. 56, 111
- [Gomes et al., 1998] Gomes, C. P., Selman, B. and Kautz, H. (1998). Boosting Combinatorial Search Through Randomization. In Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence AAAI '98/IAAI '98 pp. 431–437, American Association for Artificial Intelligence, Menlo Park, CA, USA. 52
- [Gould et al., 2004] Gould, C., Su, Z. and Devanbu, P. (2004). Static checking of dynamically generated queries in database applications. In Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on pp. 645–654, IEEE. 175
- [Grieskamp et al., 2011] Grieskamp, W., Kicillof, N., Stobie, K. and Braberman, V. (2011). Model-based Quality Assurance of Protocol Documentation: Tools and Methodology. Softw. Test. Verif. Reliab. 21, 55–71. 60

- [Haerder and Reuter, 1983] Haerder, T. and Reuter, A. (1983). Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.* 15, 287–317. 36
- [Hainaut, 2012] Hainaut, J. (2012). Bases de données - 2e éd. - Concepts, utilisation et développement: Concepts, utilisation et développement. Informatique, Dunod. 8, 25, 40, 42
- [Halfond et al., 2006] Halfond, W., Viegas, J. and Orso, A. (2006). A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA pp. 13–15, IEEE Computer Society Press. 36
- [Halfond and Orso, 2006] Halfond, W. G. J. and Orso, A. (2006). Command-Form Coverage for Testing Database Applications. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering ASE '06* pp. 69–80, IEEE Computer Society, Washington, DC, USA. 103
- [Halfond et al., 2006] Halfond, W. G. J., Orso, A. and Manolios, P. (2006). Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering SIGSOFT '06/FSE-14* pp. 175–185, ACM, New York, NY, USA. 173
- [Haller, 2009] Haller, K. (2009). White-box Testing for Database-driven Applications: A Requirements Analysis. In *Proceedings of the Second International Workshop on Testing Database Systems DBTest '09* pp. 13:1–13:6, ACM, New York, NY, USA. 102
- [Haller, 2010] Haller, K. (2010). The Test Data Challenge for Database-driven Applications. In *Proceedings of the Third International Workshop on Testing Database Systems DBTest '10* pp. 6:1–6:6, ACM, New York, NY, USA. 102
- [Hamadi et al., 2009] Hamadi, Y., Jabbour, S. and Sais, L. (2009). ManySAT: a Parallel SAT Solver. *JSAT* 6, 245–262. 52
- [Henzinger et al., 2003] Henzinger, T., Jhala, R., Majumdar, R. and Sutre, G. (2003). Software Verification with BLAST. In *Model Checking Software*, (Ball, T. and Rajamani, S., eds), vol. 2648, of *Lecture Notes in Computer Science* pp. 235–239. Springer Berlin Heidelberg. 60
- [Hickey, 2000] Hickey, T. J. (2000). Analytic Constraint Solving and Interval Arithmetic. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL '00* pp. 338–351, ACM, New York, NY, USA. 92
- [Hodges, 1997] Hodges, W. (1997). A Shorter Model Theory. Cambridge University Press, New York, NY, USA. 66, 67
- [Hooimeijer and Weimer, 2009] Hooimeijer, P. and Weimer, W. (2009). A Decision Procedure for Subset Constraints over Regular Languages. *SIGPLAN Not.* 44, 188–198. 61
- [Howden, 1975] Howden, W. E. (1975). Methodology for the generation of program test data. *Computers, IEEE Transactions on* 100, 554–560. 151

- [Huang et al., 2011] Huang, S. S., Green, T. J. and Loo, B. T. (2011). Datalog and Emerging Applications: An Interactive Tutorial. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data SIGMOD '11 pp. 1213–1216, ACM, New York, NY, USA. 41
- [Hustadt et al., 2004] Hustadt, U., Schmidt, R. A. and Georgieva, L. (2004). A Survey of Decidable First-Order Fragments and Description Logics. *Journal of Relational Methods in Computer Science* 1, 251–276. 56
- [ISO, 2011] ISO (2011). ISO/IEC 9075 : 2011 Information Technology, Database Languages, SQL. 25
- [Jackson, 2006] Jackson, D. (2006). Software Abstractions: Logic, Language, and Analysis. The MIT Press. 74, 75, 78, 79
- [Jayaraman et al., 2009] Jayaraman, K., Harvison, D., Ganesh, V. and Kiezun, A. (2009). jFuzz: A Concolic Whitebox Fuzzer for Java. In NASA Formal Methods, (Denney, E., Giannakopoulou, D. and Pasareanu, C. S., eds), vol. NASA/CP-2009-215407, of NASA Conference Proceedings pp. 121–125, NASA. 93
- [Jia and Harman, 2011] Jia, Y. and Harman, M. (2011). An Analysis and Survey of the Development of Mutation Testing. *Software Engineering, IEEE Transactions on* 37, 649–678. 95
- [Jones et al., 1993] Jones, N. D., Gomard, C. K. and Sestoft, P. (1993). Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. 173
- [Jorgensen, 2008] Jorgensen, P. C. (2008). Software Testing: A Craftsman’s Approach, Third Edition. 3 edition, Auerbach. 83
- [Jovanovic and de Moura, 2012] Jovanovic, D. and de Moura, L. M. (2012). Solving Non-linear Arithmetic. In [Reynolds et al., 2013] pp. 339–354. 56, 94, 174
- [Kaner et al., 1993] Kaner, C., Nguyen, H. Q. and Falk, J. L. (1993). Testing Computer Software. 2nd edition, Wiley & Sons, New York, NY, USA. 83
- [Kapfhammer and Soffa, 2003] Kapfhammer, G. M. and Soffa, M. L. (2003). A family of test adequacy criteria for database-driven applications. In Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering ESEC/FSE-11 pp. 98–107, ACM, New York, NY, USA. 103
- [Karmarkar, 1984] Karmarkar, N. (1984). A New Polynomial-time Algorithm for Linear Programming. In Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing STOC '84 pp. 302–311, ACM, New York, NY, USA. 59
- [Kautz et al., 1992] Kautz, H. A., Selman, B. et al. (1992). Planning as Satisfiability. In ECAI '92 Proceedings of the 10th European conference on Artificial intelligence vol. 92, pp. 359–363, John Wiley & Sons, Inc. 48

- [Khalek et al., 2008] Khalek, S. A., Elkarablieh, B., Laleye, Y. O. and Khurshid, S. (2008). Query-Aware Test Generation Using a Relational Constraint Solver. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering ASE '08 pp. 238–247, IEEE Computer Society, Washington, DC, USA. 104
- [Kiezun et al., 2009] Kiezun, A., Ganesh, V., Guo, P. J., Hooimeijer, P. and Ernst, M. D. (2009). HAMPI: a solver for string constraints. In Proceedings of the eighteenth international symposium on Software testing and analysis pp. 105–116, ACM. 61
- [Kim et al., 2012] Kim, M., Kim, Y. and Rothermel, G. (2012). A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation. In Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation ICST '12 pp. 340–349, IEEE Computer Society, Washington, DC, USA. 92
- [King, 1976] King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM* 19, 385–394. 86, 88
- [Korovin and Voronkov, 2007] Korovin, K. and Voronkov, A. (2007). Integrating Linear Arithmetic into Superposition Calculus. In Computer Science Logic, (Duparc, J. and Henzinger, T., eds), vol. 4646, of Lecture Notes in Computer Science pp. 223–237. Springer Berlin Heidelberg. 73
- [Kuznetsov et al., 2010] Kuznetsov, V., Chipounov, V. and Candea, G. (2010). Testing Closed-source Binary Device Drivers with DDT. In Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference USENIXATC'10 pp. 12–12, USENIX Association, Berkeley, CA, USA. 92
- [Lahiri and Seshia, 2004] Lahiri, S. and Seshia, S. (2004). The UCLID Decision Procedure. In Computer Aided Verification, (Alur, R. and Peled, D., eds), vol. 3114, of Lecture Notes in Computer Science pp. 475–478. Springer Berlin Heidelberg. 73
- [Leavitt, 2010] Leavitt, N. (2010). Will NoSQL Databases Live Up to Their Promise? *Computer* 43, 12–14. 43
- [Li and Csallner, 2010] Li, C. and Csallner, C. (2010). Dynamic symbolic database application testing. In Proceedings of the Third International Workshop on Testing Database Systems DBTest '10 pp. 7:1–7:6, ACM, New York, NY, USA. 102
- [Liang et al., 2014] Liang, T., Reynolds, A., Tinelli, C., Barrett, C. and Deters, M. (2014). A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In CAV, (Biere, A. and Bloem, R., eds), vol. 8559, of Lecture Notes in Computer Science pp. 646–662, Springer. 94, 174
- [lp solve, 2014] lp solve (2014). SourceForge: <http://sourceforge.net/projects/lpsolve/>. Website. 92, 100
- [Lyons, 1977] Lyons, N. R. (1977). An Automatic Data Generating System for Data Base Simulation and Testing. *SIGMIS Database* 8, 10–13. 103
- [Majumdar and Sen, 2007a] Majumdar, R. and Sen, K. (2007a). LATEST : Lazy Dynamic Test Input Generation. Technical Report UCB/EECS-2007-36 EECS Department, University of California, Berkeley. 93

- [Majumdar and Sen, 2007b] Majumdar, R. and Sen, K. (2007b). Hybrid Concolic Testing. In Proceedings of the 29th International Conference on Software Engineering ICSE '07 pp. 416–426, IEEE Computer Society, Washington, DC, USA. 93
- [Malburg and Fraser, 2011] Malburg, J. and Fraser, G. (2011). Combining search-based and constraint-based testing. In Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering ASE '11 pp. 436–439, IEEE Computer Society, Washington, DC, USA. 95
- [Malik and Zhang, 2009] Malik, S. and Zhang, L. (2009). Boolean Satisfiability from Theoretical Hardness to Practical Success. *Commun. ACM* 52, 76–82. 47, 48, 52, 53
- [Mantyla et al., 2003] Mantyla, M., Vanhanen, J. and Lassenius, C. (2003). A taxonomy and an initial empirical study of bad smells in code. In Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on pp. 381–384,. 84
- [Marchiori and Rossi, 1999] Marchiori, E. and Rossi, C. (1999). A Flipping Genetic Algorithm for Hard 3-SAT Problems. In in Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99 pp. 393–400, Morgan Kaufmann. 54
- [Marcozzi et al., 2012] Marcozzi, M., Vanhoof, W. and Hainaut, J.-L. (2012). Test input generation for database programs using relational constraints. In Proceedings of the Fifth International Workshop on Testing Database Systems DBTest '12 pp. 6:1–6:6, ACM, New York, NY, USA. 116
- [Marcozzi et al., 2013] Marcozzi, M., Vanhoof, W. and Hainaut, J.-L. (2013). A relational symbolic execution algorithm for constraint-based testing of database programs. In Source Code Analysis and Manipulation (SCAM), IEEE 13th International Working Conference on pp. 179–188, ACM. 116
- [Marcozzi et al., 2014a] Marcozzi, M., Vanhoof, W. and Hainaut, J.-L. (2014a). Towards Testing of Full-scale SQL Applications Using Relational Symbolic Execution. In Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis CSTVA 2014 pp. 12–17, ACM, New York, NY, USA. 170
- [Marcozzi et al., 2014b] Marcozzi, M., Vanhoof, W. and Hainaut, J.-L. (2014b). Testing Database Programs using Relational Symbolic Execution. Technical report University of Namur. 116
- [Marques Silva and Sakallah, 1996] Marques Silva, J. and Sakallah, K. (1996). Conflict analysis in search algorithms for satisfiability. In Tools with Artificial Intelligence, 1996., Proceedings Eighth IEEE International Conference on pp. 467–469, IEEE Computer Society Press. 52
- [McCarthy, 1960] McCarthy, J. (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 184–195. 58
- [McMinn, 2004] McMinn, P. (2004). Search-based software test data generation: a survey: Research Articles. *Softw. Test. Verif. Reliab.* 14, 105–156. 95
- [McMinn, 2011] McMinn, P. (2011). Search-Based Software Testing: Past, Present and Future. In Software Testing, Verification and Validation Workshops (ICSTW), 2011

IEEE Fourth International Conference on pp. 153–163, IEEE Computer Society Press. 95

- [Monniaux, 2010] Monniaux, D. (2010). Quantifier Elimination by Lazy Model Enumeration. In Proceedings of the 22Nd International Conference on Computer Aided Verification CAV’10 pp. 585–599, Springer-Verlag, Berlin, Heidelberg. 67
- [Moskal et al., 2008] Moskal, M., Lopuszanski, J. and Kiniry, J. R. (2008). E-matching for Fun and Profit. *Electron. Notes Theor. Comput. Sci.* 198, 19–35. 68
- [Moskewicz et al., 2001] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. and Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 38th Annual Design Automation Conference DAC ’01 pp. 530–535, ACM, New York, NY, USA. 52
- [Moura and Bjørner, 2007] Moura, L. and Bjørner, N. (2007). Efficient E-Matching for SMT Solvers. In Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction CADE-21 pp. 183–198, Springer-Verlag, Berlin, Heidelberg. 68
- [Nam et al., 1999] Nam, G.-J., Sakallah, K. A. and Rutenbar, R. A. (1999). Satisfiability-based Layout Revisited: Detailed Routing of Complex FPGAs via Search-based Boolean SAT. In Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays FPGA ’99 pp. 167–175, ACM, New York, NY, USA. 48
- [Nelson and Oppen, 1979] Nelson, G. and Oppen, D. C. (1979). Simplification by Co-operating Decision Procedures. *ACM Trans. Program. Lang. Syst.* 1, 245–257. 60, 64
- [Nelson and Oppen, 1980] Nelson, G. and Oppen, D. C. (1980). Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 356–364. 60
- [Nieuwenhuis et al., 2006] Nieuwenhuis, R., Oliveras, A. and Tinelli, C. (2006). Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM* 53, 937–977. 60, 62
- [Nieuwenhuis and Rubio, 2001] Nieuwenhuis, R. and Rubio, A. (2001). Paramodulation-Based Theorem Proving. In Handbook of Automated Reasoning, (Robinson, J. A. and Voronkov, A., eds), pp. 371–443. Elsevier and MIT Press. 69
- [O’Neil, 2008] O’Neil, E. J. (2008). Object/Relational Mapping 2008: Hibernate and the Entity Data Model (Edm). In Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data SIGMOD ’08 pp. 1351–1356, ACM, New York, NY, USA. 42
- [Oracle, 2014] Oracle (2014). JDBC Overview: <http://www.oracle.com/technetwork/java/overview-141217.html>. Website. 36, 37
- [Pablo et al., 2014] Pablo, J., Fraser, G. and Arcuri, A. (2014). Higher Coverage with Combined Search-based and Constraint-based Test Suite Generation. In ACM International Conference on Software Testing and Analysis (ISSTA) ACM. 95

- [Pan et al., 2011a] Pan, K., Wu, X. and Xie, T. (2011a). Database state generation via dynamic symbolic execution for coverage criteria. In *Proceedings of the Fourth International Workshop on Testing Database Systems* ACM, New York, NY, USA. 103
- [Pan et al., 2011b] Pan, K., Wu, X. and Xie, T. (2011b). Generating Program Inputs for Database Application Testing. In *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)* IEEE Computer Society Press. 102
- [Pan et al., 2013a] Pan, K., Wu, X. and Xie, T. (2013a). Automatic test generation for mutation testing on database applications. In *Automation of Software Test (AST)*, 8th International Workshop on pp. 111–117, IEEE Computer Society Press. 105
- [Pan et al., 2013b] Pan, K., Wu, X. and Xie, T. (2013b). Guided Test Generation for Database Applications via Synthesized Database Interactions. *ACM Transactions on Software Engineering and Methodology* 22(2), 99, 101, 102, 105, 152, 158, 159, 163, 164, 171
- [Pan et al., 2014] Pan, K., Wu, X. and Xie, T. (2014). Program-input generation for testing database applications using existing database states. *Automated Software Engineering* 22(1), 1–35. 102
- [Papadimitriou, 1981] Papadimitriou, C. H. (1981). On the Complexity of Integer Programming. *J. ACM* 28, 765–768. 56, 59
- [Paris and Harrington, 1977] Paris, J. and Harrington, L. (1977). A Mathematical Incompleteness in Peano Arithmetic. In *Handbook of Mathematical Logic* pp. 90–1133. North-Holland Pub. Co. 56
- [Pasareanu and Visser, 2009] Pasareanu, C. S. and Visser, W. (2009). A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *Int. J. Softw. Tools Technol. Transf.* 11, 339–353. 85
- [Pnueli et al., 1999] Pnueli, A., Rodeh, Y., Shtrichman, O. and Siegel, M. (1999). Deciding Equality Formulas by Small Domains Instantiations. In *Computer Aided Verification*, (Halbwachs, N. and Peled, D., eds), vol. 1633, of *Lecture Notes in Computer Science* pp. 455–469. Springer Berlin Heidelberg. 60
- [Presburger, 1931] Presburger, M. (1931). Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*. 56
- [Puri and Gu, 1996] Puri, R. and Gu, J. (1996). A BDD SAT solver for satisfiability testing: an industrial case study. *Annals of Mathematics and Artificial Intelligence* 17, 315–337. 53
- [Păsăreanu et al., 2008] Păsăreanu, C. S., Mehltz, P. C., Bushnell, D. H., Gundy-Burlet, K., Lowry, M., Person, S. and Pape, M. (2008). Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis ISSTA '08* pp. 15–26, ACM, New York, NY, USA. 92, 93

- [Ramler and Wolfmaier, 2006] Ramler, R. and Wolfmaier, K. (2006). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In *Proceedings of International Workshop on Automation of Software Test AST '06* pp. 85–91, ACM, New York, NY, USA. 83
- [Redelinghuys et al., 2012] Redelinghuys, G., Visser, W. and Geldenhuys, J. (2012). Symbolic Execution of Programs with Strings. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference SAICSIT '12* pp. 139–148, ACM, New York, NY, USA. 94, 174
- [Reynolds et al., 2013] Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M. and Barrett, C. (2013). Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *Automated Deduction – CADE-24*, (Bonacina, M., ed.), vol. 7898, of *Lecture Notes in Computer Science* pp. 377–391. Springer Berlin Heidelberg. 56, 60, 70, 193
- [Robbert and Maryanski, 1991] Robbert, M. and Maryanski, F. J. (1991). Automated Test Plan Generator for Database Applications Systems. *SIGSMALL/PC Notes* 17, 29–35. 103
- [Robinson, 1965] Robinson, J. A. (1965). A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 23–41. 69
- [Ruse et al., 2010] Ruse, M., Sarkar, T. and Basu, S. (2010). Analysis and Detection of SQL Injection Vulnerabilities via Automatic Test Case Generation of Programs. In *Applications and the Internet (SAINT)*, 2010 10th IEEE/IPSJ International Symposium on pp. 31–37, IEEE Computer Society Press. 92
- [Sakti et al., 2012] Sakti, A., Guéhéneuc, Y.-G. and Pesant, G. (2012). Boosting Search Based Testing by Using Constraint Based Testing. In *Search Based Software Engineering*, (Fraser, G. and Teixeira de Souza, J., eds), vol. 7515, of *Lecture Notes in Computer Science* pp. 213–227. Springer Berlin Heidelberg. 95
- [SAT-Association, 2014] SAT-Association (2014). SatLive: www.satlive.org. 47
- [Saxena et al., 2010] Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S. and Song, D. (2010). A Symbolic Execution Framework for JavaScript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy SP '10* pp. 513–528, IEEE Computer Society, Washington, DC, USA. 94, 174
- [Sebastiani, 2007] Sebastiani, R. (2007). Lazy Satisfiability Modulo Theories. *JSAT* 3, 141–224. 64, 65, 205
- [Selman et al., 1992] Selman, B., Levesque, H. and Mitchell, D. (1992). A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence AAAI'92* pp. 440–446, AAAI Press. 54
- [Sen and Agha, 2006] Sen, K. and Agha, G. (2006). CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th international conference on Computer Aided Verification CAV'06* pp. 419–423, Springer-Verlag, Berlin, Heidelberg. 92, 100
- [Sen et al., 2005] Sen, K., Marinov, D. and Agha, G. (2005). CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes* 30, 263–272. 89, 90

- [Shannon et al., 2009] Shannon, D., Ghosh, I., Rajan, S. and Khurshid, S. (2009). Efficient Symbolic Execution of Strings for Validating Web Applications. In Proceedings of the 2Nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009) DEFECTS '09 pp. 22–26, ACM, New York, NY, USA. 94, 174
- [Shostak, 1978] Shostak, R. E. (1978). An Algorithm for Reasoning About Equality. *Commun. ACM* 21, 583–585. 60
- [Shostak, 1984] Shostak, R. E. (1984). Deciding Combinations of Theories. *J. ACM* 31, 1–12. 207
- [Siddiqui and Khurshid, 2010] Siddiqui, J. and Khurshid, S. (2010). ParSym: Parallel symbolic execution. In Software Technology and Engineering (ICSTE), 2010 2nd International Conference on vol. 1, pp. V1–405–V1–409, IEEE Computer Society Press. 93
- [Silberschatz et al., 2010] Silberschatz, A., Korth, H. and Sudarshan, S. (2010). Database System Concepts. Connect, learn, succeed, McGraw-Hill Education. 40, 41
- [Silva, 1999] Silva, J. a. P. M. (1999). The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence EPIA '99 pp. 62–74, Springer-Verlag, London, UK, UK. 52
- [Spears, 1996] Spears, W. (1996). A NN algorithm for Boolean satisfiability problems. In Neural Networks, 1996., IEEE International Conference on vol. 2, pp. 1121–1126 vol.2, IEEE Computer Society Press. 54
- [Spears, 1993] Spears, W. M. (1993). Simulated annealing for hard satisfiability problems. Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge 26, 533–558. 54
- [Staats and Păsăreanu, 2010] Staats, M. and Păsăreanu, C. (2010). Parallel Symbolic Execution for Structural Test Generation. In Proceedings of the 19th International Symposium on Software Testing and Analysis ISSTA '10 pp. 183–194, ACM, New York, NY, USA. 93
- [Stalmarck, 1994] Stalmarck, G. (1994). System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula. US Patent 5,276,897. 53
- [Stonebraker, 2010] Stonebraker, M. (2010). SQL Databases V. NoSQL Databases. *Commun. ACM* 53, 10–11. 43
- [Stonebraker, 2012] Stonebraker, M. (2012). New opportunities for New SQL. *Commun. ACM* 55, 10–11. 43
- [Stonebraker and Hellerstein, 1998] Stonebraker, M. and Hellerstein, J. M., eds (1998). Readings in Database Systems, Third Edition. Morgan Kaufmann. 41

- [Stonebraker et al., 2007] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N. and Helland, P. (2007). The End of an Architectural Era: (It’s Time for a Complete Rewrite). In Proceedings of the 33rd International Conference on Very Large Data Bases VLDB ’07 pp. 1150–1160, VLDB Endowment. 43
- [Su and Wassermann, 2006] Su, Z. and Wassermann, G. (2006). The essence of command injection attacks in web applications. SIGPLAN Not. *41*, 372–382. 173
- [Suárez-Cabal and Tuya, 2004] Suárez-Cabal, M. J. and Tuya, J. (2004). Using an SQL coverage measurement for testing database applications. In Proceedings of the 12th ACM SIGSOFT twelfth International Symposium on Foundations of Software Engineering SIGSOFT ’04/FSE-12 pp. 253–262, ACM, New York, NY, USA. 103
- [Suárez-Cabal and Tuya, 2009] Suárez-Cabal, M. J. and Tuya, J. (2009). Structural coverage criteria for testing SQL queries. Journal of Universal Computer Science *15*, 584–619. 103
- [Taneja et al., 2010] Taneja, K., Zhang, Y. and Xie, T. (2010). MODA: Automated test generation for database applications via mock objects. In In Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE), short paper ACM. 99, 105
- [Tateishi et al., 2013] Tateishi, T., Pistoia, M. and Tripp, O. (2013). Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic. ACM Trans. Softw. Eng. Methodol. *22*, 33:1–33:33. 94, 174
- [Tillmann and De Halleux, 2008] Tillmann, N. and De Halleux, J. (2008). Pex: white box test generation for .NET. In Proceedings of the 2nd international conference on Tests and proofs TAP’08 pp. 134–153, Springer-Verlag, Berlin, Heidelberg. 93, 100, 102
- [Tinelli and Zarba, 2005] Tinelli, C. and Zarba, C. G. (2005). Combining Nonstably Infinite Theories. J. Autom. Reason. *34*, 209–238. 60
- [Torlak and Jackson, 2007] Torlak, E. and Jackson, D. (2007). Kodkod: A Relational Model Finder. In TACAS, (Grumberg, O. and Huth, M., eds), vol. 4424, of Lecture Notes in Computer Science pp. 632–647, Springer. 78
- [Torlak et al., 2013] Torlak, E., Taghdiri, M., Dennis, G. and Near, J. P. (2013). Applications and extensions of Alloy: past, present and future. Mathematical Structures in Computer Science *23*, 915–933. 74
- [Tseitin, 1983] Tseitin, G. S. (1983). On the Complexity of Derivation in Propositional Calculus. In Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970, (Siekman, J. and Wrightson, G., eds), pp. 466–483. Springer Berlin, Heidelberg. 49
- [Turing, 1936] Turing, A. (1936). On computable numbers with an application to the “Entscheidungsproblem”. Proceeding of the London Mathematical Society -. 56
- [Tuya et al., 2006] Tuya, J., Suarez-Cabal, M. J. and de la Riva, C. (2006). SQLMutation: A Tool to Generate Mutants of SQL Database Queries. In Proceedings of the Second Workshop on Mutation Analysis MUTATION ’06 pp. 1–, IEEE Computer Society, Washington, DC, USA. 105

- [Tuya et al., 2010] Tuya, J., Suárez-Cabal, M. J. and de la Riva, C. (2010). Full Predicate Coverage for Testing SQL Database Queries. *Softw. Test. Verif. Reliab.* 20, 237–288. 103, 104
- [Veanes et al., 2010] Veanes, M., De Halleux, P. and Tillmann, N. (2010). Rex: Symbolic regular expression explorer. In *Software Testing, Verification and Validation (ICST)*, 2010 Third International Conference on pp. 498–507, IEEE. 61
- [Veanes et al., 2009] Veanes, M., Grigorenko, P., Halleux, P. and Tillmann, N. (2009). Symbolic Query Exploration. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering ICFEM '09* pp. 49–68, Springer-Verlag, Berlin, Heidelberg. 104, 164
- [Veanes et al., 2010] Veanes, M., Tillmann, N. and de Halleux, J. (2010). Qex: Symbolic SQL Query Explorer. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning LPAR'10* pp. 425–446, Springer-Verlag, Berlin, Heidelberg. 104, 111, 164
- [Willmor and Embury, 2005] Willmor, D. and Embury, S. M. (2005). Exploring test adequacy for database systems. In *3rd UK Software Testing Research workshop* pp. 123–133, Software Testing Research workshop. 103
- [Willmor and Embury, 2006] Willmor, D. and Embury, S. M. (2006). An intensional approach to the specification of test cases for database applications. In *Proceedings of the 28th international conference on Software engineering ICSE '06* pp. 102–111, ACM, New York, NY, USA. 104
- [Xie, 2009] Xie, T. (2009). Improving automation in developer testing: State of the practice. Technical report North Carolina State University. 93
- [Xu et al., 2010] Xu, Z., Kim, Y., Kim, M., Rothermel, G. and Cohen, M. B. (2010). Directed test suite augmentation: techniques and tradeoffs. In *SIGSOFT FSE*, (Roman, G.-C. and Sullivan, K. J., eds), pp. 257–266, ACM. 92
- [Zamfir and Candea, 2010] Zamfir, C. and Candea, G. (2010). Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems EuroSys '10* pp. 321–334, ACM, New York, NY, USA. 92
- [Zhang et al., 2001] Zhang, J., Xu, C. and Cheung, S. C. (2001). Automatic Generation of Database Instances for White-box Testing. In *Proceedings of the 25th International Computer Software and Applications Conference COMPSAC '01* pp. 161–165, IEEE Computer Society, Washington, DC, USA. 103
- [Zheng et al., 2013] Zheng, Y., Zhang, X. and Ganesh, V. (2013). Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering ESEC/FSE 2013* pp. 114–124, ACM, New York, NY, USA. 94, 174
- [Zhou and Frankl, 2009] Zhou, C. and Frankl, P. (2009). Mutation Testing for Java Database Applications. In *Proceedings of the International Conference on Software Testing Verification and Validation ICST '09* pp. 396–405, IEEE Comp. Soc., Washington, DC, USA. 105

- [Zhou and Frankl, 2012] Zhou, C. and Frankl, P. (2012). Empirical Studies on Test Effectiveness for Database Applications. In Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on pp. 61–70, IEEE Computer Society Press. 103
- [Zhu et al., 1997] Zhu, H., Hall, P. A. V. and May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Comput. Surv.* 29, 366–427. 85, 95, 102, 151

Part IV

Appendices

Appendix A

Nelson-Oppen Combination

A.1 Convex Σ -theories

The general version of the Nelson-Oppen algorithm is exponential in time. If the considered Σ -theories are *convex*, a simplified, polynomial in time, version of the Nelson-Oppen solver combination algorithm can be used:

Definition A.1. *A Σ -theory is **convex** if, for every satisfiable theory made of formulas from this Σ -theory, there exists at least one model satisfying the theory, where the uninterpreted constants not implied to be equal by the theory have distinct semantic values.*

For example [de Moura and Bjørner, 2008], linear integer arithmetic is not convex, as the following satisfiable theory, involving constant symbols a , b and c :

$$(7 \leq a \leq 8) \wedge (7 \leq b \leq 8) \wedge (7 \leq c \leq 8)$$

does not imply any equality between a , b and c , but admits no model where all the three constants are different.

The main principles of both the convex and non-convex version of the Nelson-Oppen algorithm are similar. An illustrative example of the convex version is provided in this appendix. A detailed discussion of both the generalized and simplified algorithm, together with a presentation of their theoretical background and of the existing extended and alternate approaches can be found e.g. in [Barrett et al., 2009, Sebastiani, 2007].

A.2 Convex Nelson-Oppen Combination

We describe now how the Nelson-Oppen method, in its simplest polynomial form, works. In order to do so and without loss of generality, we demonstrate the integration of a Σ -theory solver for uninterpreted functions (UF) with a Σ -theory solver for linear real arithmetic (LRA) to solve the theory composed of the following formula, modulo uninterpreted functions and linear real arithmetic:

$$(f(x) = 1 + f(0) + f(x - 1)) \wedge (f(x) - f(f(0)) = 0) \wedge (f(0) + f(x) = 1 + f(x - 1))$$

where “=” is the set equality predicate, “ f ” an uninterpreted function from a set to itself, “0”, “1”, “+”, “−” the classical symbols of linear real arithmetic and “ x ” an uninterpreted constant. The two considered Σ -theories (LRA and UF) are disjoint, stably-infinite and

convex. Used within the DPLL(T) algorithm, the demonstrated solver makes it possible to create an SMT solver for the QF_UFLRA logic.

First, the Nelson-Oppen solver isolates the Σ -theory literals appearing in the theory, which must all be satisfied for the theory to be true. In the considered example, we have the three following literals:

$$\begin{cases} f(x) = 1 + f(0) + f(x - 1) \\ f(x) - f(f(0)) = 0 \\ f(0) + f(x) = 1 + f(x - 1) \end{cases}$$

Secondly, the Nelson-Oppen solver purifies each of the obtained literals. Purifying a literal means to replace it by an equisatisfiable set of *pure* literals, i.e. literals that belong to a single Σ -theory. The purification process simply recursively replaces every alien term t by a new uninterpreted constant c and adds the literal $c = t$, until all literals are pure. For example, for the first literal considered here:

$$f(x) = 1 + f(0) + f(x - 1)$$

the first purification step produce the following equisatisfiable set of literals:

$$\begin{cases} e_1 = 1 + e_2 + e_3 & (\text{Pure LRA}) \\ e_1 = f(x) & (\text{Pure UF}) \\ e_2 = f(0) & (\text{Mixed}) \\ e_3 = f(x - 1) & (\text{Mixed}) \end{cases}$$

As the two last literals are not pure, the process must be repeated recursively leading to:

$$\begin{cases} e_1 = 1 + e_2 + e_3 & (\text{Pure LRA}) \\ e_1 = f(x) & (\text{Pure UF}) \\ e_2 = f(e_4) & (\text{Pure UF}) \\ e_3 = f(e_5) & (\text{Pure UF}) \\ e_4 = 0 & (\text{Pure LRA}) \\ e_5 = x - 1 & (\text{Pure LRA}) \end{cases}$$

where all literals are pure. The second and third original literals $f(x) - f(f(0)) = 0$ and $f(0) + f(x) = 1 + f(x - 1)$ can be purified in the same way, providing respectively the two following sets of pure literals:

$$\begin{cases} e_6 - e_7 = 0 & (\text{Pure LRA}) \\ e_6 = f(x) & (\text{Pure UF}) \\ e_7 = f(f(e_8)) & (\text{Pure UF}) \\ e_8 = 0 & (\text{Pure LRA}) \end{cases}$$

$$\begin{cases} e_9 + e_{10} = 1 + e_{11} & (\text{Pure LRA}) \\ e_9 = f(e_{12}) & (\text{Pure UF}) \\ e_{10} = f(x) & (\text{Pure UF}) \\ e_{11} = f(e_{13}) & (\text{Pure UF}) \\ e_{12} = 0 & (\text{Pure LRA}) \\ e_{13} = x - 1 & (\text{Pure LRA}) \end{cases}$$

Thirdly, the Nelson-Oppen solver groups each set of pure literals belonging to a single Σ -theory together. In our example we have:

Pure LRA	Pure UF
$e_1 = 1 + e_2 + e_3$	$e_1 = f(x)$
$e_4 = 0$	$e_2 = f(e_4)$
$e_5 = x - 1$	$e_3 = f(e_5)$
$e_6 - e_7 = 0$	$e_6 = f(x)$
$e_8 = 0$	$e_7 = f(f(e_8))$
$e_9 + e_{10} = 1 + e_{11}$	$e_9 = f(e_{12})$
$e_{12} = 0$	$e_{10} = f(x)$
$e_{13} = x - 1$	$e_{11} = f(e_{13})$

The conjunction of all the pure literals in one group constitute a theory made of a single formula from the group's Σ -theory. A satisfying model can be found for such a theory, modulo the Σ -theory associated to the group, using the available solver for this Σ -theory. As the Σ -theories are disjoint and as the obtained groups share some uninterpreted constants, finding a satisfying model for the original instance to be solved by the Nelson-Oppen solver is equivalent to finding a satisfying model for the theory of each group, so that, if two of the shared constants are equal in all the satisfying models of one of the theories, the satisfying models of the other theories in which these constants are not equal must be rejected. The Nelson-Oppen method is based on this principle and supposes that each of the Σ -theory solver to be combined can deduce all the constant equalities that must be true in all the models satisfying their input theory (see e.g. [Shostak, 1984] for a common approach to build such solvers). Fourthly, the Nelson-Oppen method applies thus the following steps:

1. Select one group of pure literals and solve it using the adequate Σ -theory solver.
2.
 - **If the theory is satisfiable**, obtain the list of constant equalities true in every model satisfying the theory and add them as new literals to the other groups sharing the involved constants. Restart at step 1 over another group of literals.
 - **else**, stop, as the original instance to be solved by the Nelson-Oppen is unsatisfiable as well.
3. If the theories of all groups are satisfiable and that no more shared constant equalities can be obtained from any group, the original instance to be solved by the Nelson-Oppen solver is satisfiable. A satisfying model can be obtained by unifying the satisfying models found for the theories of all the groups.

In the considered example, the Nelson-Oppen solver can, for instance, select the group of pure LRA literals and provide it to the LRA Σ -theory solver. The resulting theory is satisfiable. As e_4 , e_8 and e_{12} must all equal 0, these three uninterpreted constants will be equal in any model satisfying the resulting theory. Similarly, e_5 and e_{13} must all equal $x - 1$ and will be equal in any model satisfying the theory. Finally, as $e_6 - e_7 = 0$ must be true, e_6 and e_7 must also be equal in any model satisfying the theory. These equalities can be added to the group of pure UF literals, leading to the following updated group, to be provided to the UF Σ -theory solver:

$$\begin{aligned}
& \text{Pure UF} \\
& e_1 = f(x) \\
& e_2 = f(e_4) \\
& e_3 = f(e_5) \\
& e_6 = f(x) \\
& e_6 = f(f(e_4)) \\
& e_9 = f(e_4) \\
& e_{10} = f(x) \\
& e_{11} = f(e_5) \\
& e_7 = e_6 \\
& e_8 = e_4 \\
& e_{12} = e_4 \\
& e_{13} = e_5
\end{aligned}$$

The resulting theory is satisfiable and generates a new set of constant equalities (like e.g. $e_2 = e_9$ as these two constants are equal to $f(e_4)$) to be added to the LRA group. The new LRA group is provided to the LRA Σ -theory solver and the process is repeated. Finally, the two following sets of pure literals are obtained. They are both satisfiable and no more equality can be deduced:

Pure LRA	Pure UF
$e_3 = -1$	$e_1 = f(x)$
$e_1 = 0$	$e_1 = f(e_1)$
$e_5 = x - 1$	$e_3 = f(e_5)$
$e_2 = e_1$	$e_2 = e_1$
$e_4 = e_1$	$e_4 = e_1$
$e_6 = e_1$	$e_6 = e_1$
$e_7 = e_1$	$e_7 = e_1$
$e_8 = e_1$	$e_8 = e_1$
$e_9 = e_1$	$e_9 = e_1$
$e_{10} = e_1$	$e_{10} = e_1$
$e_{11} = e_3$	$e_{11} = e_3$
$e_{12} = e_1$	$e_{12} = e_1$
$e_{13} = e_5$	$e_{13} = e_5$

The original theory to be solved by the Nelson-Oppen solver is thus satisfiable. A satisfying model for the LRA group is:

$$\{x = 10, e_1 = e_2 = e_4 = e_6 = e_7 = e_8 = e_9 = e_{10} = e_{12} = 0, e_3 = e_{11} = -1, e_5 = e_{13} = 9\}$$

A satisfying model for the UF group is

The function f and all the constants are defined over a set S .

s_1, s_2, s_3, s_4 and s_5 are values taken from this set.

The function f is defined as:

$$f : S \rightarrow S : f(s) = \begin{cases} s_2 & \text{if } s = s_1 \vee s = s_2 \\ s_3 & \text{if } s = s_4 \\ s_5 & \text{otherwise} \end{cases}$$

The constants are given the following values:

$$\{x = s_1, e_1 = e_2 = e_4 = e_6 = e_7 = e_8 = e_9 = e_{10} = e_{12} = s_2, e_3 = e_{11} = s_3, e_5 = e_{13} = s_4\}$$

The two models can be combined by unifying the set S to the set of integers, s_1 to 10, s_2 to 0, s_3 to -1 , s_4 to 9 and s_5 to any integer value. By restricting the unified model to the function f and constant x , one can obtain a satisfying model for the original theory, modulo linear real arithmetic and uninterpreted functions:

$$f : \mathbb{Z} \rightarrow \mathbb{Z} : f(z) = \begin{cases} 0 & \text{if } z = 0 \vee z = 10 \\ -1 & \text{if } z = 9 \\ 666 & \text{otherwise} \end{cases}$$

$$\{x = 10\}$$

Appendix B

Code Samples Used for Evaluation

B.1 SimpleDB Samples

B.1.1 SimpleDB Language Unit Testing

A Integers

```
1 int test (Connection con,Scanner in) throws SQLException {  
2     int input1 = in.nextInt();  
3     int input2 = in.nextInt();  
4     int internalValue = 3;  
5     int computation = (((internalValue + input1) + (3 + internalValue)) - 3) ;  
6     computation = (((computation - (2 + internalValue))-input1)-input1)-input1);  
7     computation = (((computation + 1) + internalValue) + input2);  
8     return computation;  
9 }
```

B Lists

```
1 int test (Connection con,Scanner in,List< Integer> inputList) throws SQLException {  
2     List< Integer> internalValue = null;  
3     internalValue = new ArrayList< Integer> ();  
4     internalValue.add((3+ 2));  
5     internalValue.add((3-2));  
6     internalValue.remove(0);  
7     int returnValue = inputList.get(internalValue.get((internalValue.size ()-1)));  
8     inputList = internalValue;  
9     returnValue = (returnValue + inputList.size ());  
10    return returnValue;  
11 }
```

C If and While

```
1 int test (Connection con,Scanner in) throws SQLException {
2     if (true) {};
3     if (false) {};
4     if (true) {} else { };
5     if (false) {} else {};
6     while (false) {};
7     int i = 0;
8     while ((i< 1)) {
9         i = (i + 1);
10    };
11    i = 0;
12    if (true) {
13        while ((i< 1)) {
14            i = (i + 1);
15            if ((i==1)) {
16                while (true) {
17                    return i;
18                };
19            };
20        };
21    };
22    return i;
23 }
```

D Conditions

```
1 int test (Connection con,Scanner in) throws SQLException {
2     if ((false | true)) {};
3     if ((true & false)) {};
4     if (((true & false) | true)) {} else {};
5     if (((true & false) | false)) {} else {};
6     while ((! ( (true & false) | true) ) ) {};
7     int i = 0;
8     while ((i< 1)) {
9         i = (i + 1);
10    };
11    i = 0;
12    List< Integer> liste = null;
13    List< Integer> liste2 = new ArrayList< Integer> ();
14    if (true) {
15        while ((i< 1)) {
16            i = (i + 1);
17            if ((i==1)) {
18                while (((liste == null) & (2 > (i+ liste2.size ()))) {
19                    return i;
20                };
21            };
22        };
23    };
24 }
```

```

21     };
22     };
23 };
24     return i;
25 }

```

E Database Schema for Select, Insert and Transactions

```

1  CREATE TABLE t1 (
2  idt1 INTEGER NOT NULL,
3  fieldt1 INTEGER NOT NULL,
4  CONSTRAINT t1PK PRIMARY KEY (idt1),
5  CHECK(idt1 > 0)
6  );
7
8  CREATE TABLE t2 (
9  idt2 INTEGER NOT NULL,
10 fieldt2 INTEGER NOT NULL,
11 CONSTRAINT t2PK PRIMARY KEY (idt2),
12 CONSTRAINT t2FK FOREIGN KEY (fieldt2) REFERENCES t1(idt1),
13 CHECK(idt2 > 0)
14 );

```

F Database Schema for Update and Delete

```

1  CREATE TABLE t1 (
2  idt1 INTEGER NOT NULL,
3  fieldt1 INTEGER NOT NULL,
4  CONSTRAINT t1PK PRIMARY KEY (idt1)
5  );
6
7  CREATE TABLE t2 (
8  idt2 INTEGER NOT NULL,
9  fieldt2 INTEGER NOT NULL,
10 CONSTRAINT t2PK PRIMARY KEY (idt2),
11 CONSTRAINT t2FK FOREIGN KEY (fieldt2) REFERENCES t1(idt1),
12 CHECK(idt2 < 100)
13 );

```

G Select

```

1  int test (Connection con,Scanner in) throws SQLException {
2      con.createStatement().execute("DELETE FROM t2");
3      con.createStatement().execute("INSERT INTO t2 VALUES (2,2)");
4      con.createStatement().execute("INSERT INTO t2 VALUES (3,1)");
5      con.createStatement().execute("INSERT INTO t2 VALUES (1,3)");

```

```

6  ResultSet result = con.createStatement().executeQuery(
7      "SELECT idt2, fieldt2 FROM t2");
8  int value = 0;
9  int memory = 0;
10 while (result.next()) {
11     result.next();
12     result.next();
13     value = (value + result.getInt("fieldt2 "));
14     int memory = result.getInt("idt2");
15 };
16 result = con.createStatement().executeQuery(
17     "SELECT idt2 FROM t2 WHERE (idt2 = "+ (memory)+ ")");
18 while (result.next()) {
19     value = (value + result.getInt("idt2"));
20 };
21 int two = 2;
22 result = con.createStatement().executeQuery(
23     "SELECT idt2 FROM t2
24     WHERE ((idt2=fieldt2) OR ((idt2=2) AND (fieldt2="+ (two)+ ")))");
25 while (result.next()) {
26     value = (value + result.getInt("idt2"));
27 };
28 return value;
29 }

```

H Insert

```

1  void test (Connection con,Scanner in) throws SQLException {
2      con.createStatement().execute("DELETE FROM t2");
3      int value = 3;
4      con.createStatement().execute("INSERT INTO t2 VALUES (" + ((value-1))+ ",2)");
5      try {
6          con.createStatement().execute("INSERT INTO t2 VALUES (2,5)");
7      } catch (SQLException e) {};
8      try {
9          con.createStatement().execute("INSERT INTO t2 VALUES (0,5)");
10     } catch (SQLException e) {};
11     con.createStatement().execute("DELETE FROM t2");
12     con.createStatement().execute("DELETE FROM t1");
13     try {
14         con.createStatement().execute("INSERT INTO t2 VALUES (2,5)");
15     } catch (SQLException e) {};
16 }

```

I Update

```

1  void test (Connection con,Scanner in) throws SQLException {

```

```

2  con.createStatement().execute("DELETE FROM t1 WHERE (idt1 = 44)");
3  con.createStatement().execute("INSERT INTO t2 VALUES (2,1)");
4  con.createStatement().execute("INSERT INTO t2 VALUES (3,2)");
5  con.createStatement().execute("INSERT INTO t2 VALUES (1,0)");
6  int value = 1;
7  con.createStatement().execute(
8      "UPDATE t2 SET idt2 = (idt2 - "+ (value)+ ") WHERE (idt2 = (1 + fieldt2))");
9  con.createStatement().execute("UPDATE t2 SET fieldt2 = 3 WHERE (idt2 = 0)");
10 con.createStatement().execute("UPDATE t1 SET idt1 = 5 WHERE (idt1 = 4)");
11 try {
12     con.createStatement().execute("UPDATE t2 SET idt2=12 WHERE (idt2=fieldt2)");
13 } catch (SQLException e) {};
14 try {
15     con.createStatement().execute("UPDATE t1 SET idt1=15 WHERE (idt1 = 3)");
16 } catch (SQLException e) {};
17 try {
18     con.createStatement().execute("UPDATE t2 SET fieldt2 = 44 WHERE (idt2 = 2)");
19 } catch (SQLException e) {};
20 try {
21     con.createStatement().execute("UPDATE t2 SET idt2 = 101 WHERE (idt2 = 1)");
22 } catch (SQLException e) {};
23 }

```

J Delete

```

1  void test (Connection con,Scanner in) throws SQLException {
2      int value = 1;
3      con.createStatement().execute(
4          "DELETE FROM t1 WHERE (idt1 = (fieldt1 - "+ (value)+ "))");
5      con.createStatement().execute("INSERT INTO t2 VALUES (2,1)");
6      try {
7          con.createStatement().execute(
8              "DELETE FROM t1 WHERE (idt1 = "+ (value)+ ")");
9      } catch (SQLException e) {};
10 }

```

K Transactions

```

1  int test (Connection con,Scanner in,List< Integer> list) throws SQLException {
2      con.commit();
3      con.createStatement().execute("INSERT INTO t2 VALUES (2,1)");
4      if ((list == null)) {
5          con.commit();
6      } else {
7          con.rollback();
8      };
9      ResultSet result = con.createStatement().executeQuery(

```

```
10     "SELECT idt2 FROM t2 WHERE (idt2 = 2)");
11     int value = 0;
12     while (result.next()) {
13         value = (value + result.getInt("idt2"));
14     };
15     return value;
16 }
```

B.1.2 Realistic OLTP Methods

A Handling Data

```
1 CREATE TABLE data (  
2   id INTEGER NOT NULL,  
3   pointer INTEGER NOT NULL,  
4   checksum INTEGER NOT NULL,  
5   CONSTRAINT dataPK PRIMARY KEY (id),  
6   CHECK(id > 0),  
7   CHECK(id < 6),  
8   CHECK(pointer > 0),  
9   CHECK(pointer < 6),  
10  CHECK(checksum > 1),  
11  CHECK(checksum < 11)  
12 );
```

```
1 void test (Connection con,Scanner in,  
2   List< Integer>  orderedList,List< Integer>  inversedList) throws SQLException {  
3   con.commit();  
4   List< Integer>  doubleList= null;  
5   int computedLength = 0;  
6   int inputOK = 0;  
7   if (((!( orderedList==null)) & (!(inversedList==null)))) {  
8     if ((orderedList.size ()==inversedList.size())) {  
9       doubleList = new ArrayList< Integer> ();  
10      while (((computedLength< orderedList.size())  
11        & (computedLength< inversedList.size())) {  
12        computedLength = (computedLength + 1);  
13        int doubleElement = in.nextInt();  
14        doubleList.add(doubleElement);  
15      };  
16      inputOK = 1;  
17    };  
18  };  
19  if ((inputOK == 1)) {  
20    int i = 0;  
21    while (((i< computedLength) & (inputOK==1))) {  
22      if (((!( orderedList.get(i)==inversedList.get(((computedLength-1)-i))))  
23        | (!(2 *orderedList.get(i))==doubleList.get(i)))) {  
24        inputOK = 0;  
25      };  
26      i = (i + 1);  
27    };  
28    if ((inputOK == 1)) {  
29      i = 0;  
30      while ((i< computedLength)) {  
31        con.createStatement().execute(  
32          "INSERT INTO data
```

```

33         VALUES (" + (orderedList.get(i)) + ","
34                 + (invertedList.get(i)) + ","
35                 + (doubleList.get(i)) + ")");
36         i = (i + 1);
37     };
38 };
39 };
40 con.commit();
41 }

```

B Clients and Products

```

1  CREATE TABLE client (
2  name INTEGER NOT NULL,
3  age INTEGER NOT NULL,
4  purchases INTEGER NOT NULL,
5  expenses INTEGER NOT NULL,
6  CONSTRAINT clientPK PRIMARY KEY (name),
7  CHECK(name > 0),
8  CHECK(name < 5),
9  CHECK(age > 0));
10
11 CREATE TABLE prospect (
12 prospectName INTEGER NOT NULL,
13 prospectLevel INTEGER NOT NULL,
14 CONSTRAINT prospectPK PRIMARY KEY (prospectName),
15 CONSTRAINT prispectToClientFK FOREIGN KEY (prospectName)
16 REFERENCES client (name),
17 CHECK(prospectLevel > 0));
18
19 CREATE TABLE product (
20 id INTEGER NOT NULL,
21 productName INTEGER NOT NULL,
22 price INTEGER NOT NULL,
23 boughtQuantity INTEGER NOT NULL,
24 CONSTRAINT productPK PRIMARY KEY (id),
25 CHECK(price > 0));
26
27 CREATE TABLE purchase (
28 purchaseId INTEGER NOT NULL,
29 cli INTEGER NOT NULL,
30 pro INTEGER NOT NULL,
31 CONSTRAINT purchasePK PRIMARY KEY (purchaseId),
32 CONSTRAINT purchaseToClientFK FOREIGN KEY (cli)
33 REFERENCES client (name),
34 CONSTRAINT purchaseToProductFK FOREIGN KEY (pro)
35 REFERENCES product (id));

```

```

1  void test (Connection con,Scanner in) throws SQLException {
2  con.commit();
3  con.createStatement().execute("INSERT INTO client VALUES (3,4,3,5)");
4  con.createStatement().execute("INSERT INTO purchase VALUES (1,2,1)");
5  con.createStatement().execute("INSERT INTO purchase VALUES (2,1,1)");
6  con.createStatement().execute("INSERT INTO purchase VALUES (3,1,2)");
7  con.createStatement().execute("UPDATE client SET expenses=0");
8  con.createStatement().execute("UPDATE client SET purchases=0");
9  con.createStatement().execute("UPDATE product SET boughtQuantity=0");
10 ResultSet result = con.createStatement().executeQuery(
11     "SELECT purchaseId, cli, pro FROM purchase");
12 while (result.next()) {
13     con.createStatement().execute(
14         "UPDATE product SET boughtQuantity = (boughtQuantity + 1)
15         WHERE (id = "+ (result.getInt("pro"))+ ")");
16     ResultSet productPrice = con.createStatement().executeQuery(
17         "SELECT price FROM product WHERE (id = "+ (result.getInt("pro"))+ ")");
18     productPrice.next();
19     con.createStatement().execute(
20         "UPDATE client SET expenses = (expenses + "+ (productPrice.getInt("price"))+ ")
21         WHERE (name = "+ (result.getInt("cli"))+ ")");
22     con.createStatement().execute(
23         "UPDATE client SET purchases = (purchases + 1)
24         WHERE (name = "+ (result.getInt("cli"))+ ")");
25 };
26 con.createStatement().execute("DELETE FROM product
27     WHERE (boughtQuantity=0)");
28 con.createStatement().execute(
29     "UPDATE client SET name = (name+ 1)
30     WHERE ((expenses = 0) AND (purchases = 0))");
31 result = con.createStatement().executeQuery(
32     "SELECT name FROM client
33     WHERE ((purchases < 2) AND (expenses < 2))");
34 while (result.next()) {
35     con.createStatement().execute(
36         "INSERT INTO prospect VALUES (" + (result.getInt("name"))+ ",1)");
37 };
38 con.createStatement().execute("UPDATE purchase SET pro = 3
39     WHERE (pro = 2)");
40 con.createStatement().execute("DELETE FROM purchase
41     WHERE (pro = 2)");
42 con.commit();
43 }

```

C Play Catalogs

```
1 CREATE TABLE author (  
2 name INTEGER NOT NULL,  
3 totalAudience INTEGER NOT NULL,  
4 CONSTRAINT authorPK PRIMARY KEY (name),  
5 CHECK(totalAudience > 0)  
6 );  
7  
8 CREATE TABLE play (  
9 title INTEGER NOT NULL,  
10 authorName INTEGER NOT NULL,  
11 audience INTEGER NOT NULL,  
12 CONSTRAINT playPK PRIMARY KEY (title),  
13 CONSTRAINT playToAuthorFK FOREIGN KEY (authorName)  
14 REFERENCES author(name),  
15 CHECK(audience > 0)  
16 );
```

```
1 void test (Connection con,Scanner in,List< Integer> newAuthors,  
2 List< Integer> newPlaysTitle,List< Integer> newPlaysAudience,  
3 List< Integer> deadAuthors) throws SQLException {  
4 con.commit();  
5 List< Integer> newPlaysAuthor = newAuthors;  
6 int error = 0;  
7 if (!(newAuthors==null)) {  
8 int i = 0;  
9 while (((error==0) & (i< newAuthors.size())) {  
10 try {  
11 con.createStatement().execute("INSERT INTO author  
12 VALUES (" + (newPlaysAuthor.get(i)) + ",1)");  
13 } catch (SQLException e) {  
14 error = 1;  
15 };  
16 i = (i + 1);  
17 };  
18 } else {  
19 error = 1;  
20 };  
21 if (((!(deadAuthors==null)) & (error==0))) {  
22 int i = 0;  
23 while (((error==0) & (i< deadAuthors.size())) {  
24 try {  
25 con.createStatement().execute("DELETE FROM author  
26 WHERE (name=" + (deadAuthors.get(i)) + ")");  
27 } catch (SQLException e) {  
28 error = 1;  
29 };
```

```

30     i = (i + 1);
31 };
32 } else {
33     error = 1;
34 };
35 if ((error==0)) {
36     con.commit();
37 } else {
38     con.rollback();
39 };
40 error = 0;
41 if (((!((( newPlaysTitle==null) | (newPlaysAuthor==null))
42 | (newPlaysAudience==null))) & ((newPlaysTitle.size()==newPlaysAuthor.size())
43 & (newPlaysAudience.size()==newPlaysTitle.size())))) {
44     int i = 0;
45     while (((error==0) & (i< newPlaysTitle.size())) {
46         try {
47             con.createStatement().execute("INSERT INTO play
48                 VALUES (" + (newPlaysTitle.get(i)) + ",
49                     " + (newPlaysAuthor.get(i)) + ",
50                     " + (newPlaysAudience.get(i)) + ")");
51         } catch (SQLException e) {
52             error = 1;
53         };
54         try {
55             con.createStatement().execute("UPDATE author
56                 SET totalAudience = (totalAudience + " + (newPlaysAudience.get(i)) + ")
57                 WHERE (name = " + (newPlaysAuthor.get(i)) + ")");
58         } catch (SQLException e) {
59             error = 1;
60         };
61         i = (i + 1);
62     };
63 } else {
64     error = 1;
65 };
66 if ((error==0)) {
67     con.commit();
68 } else {
69     con.rollback();
70 }; }

```

B.2 Samples Extracted from OpenSource Software

B.2.1 UnixUsage

A Database Schema

```
1 CREATE TABLE COURSE_INFO (
2 COURSE_ID INTEGER NOT NULL,
3 COURSE_NAME VARCHAR(50),
4 OFFERED_DEPT INTEGER,
5 GRADUATE_LEVEL SMALLINT,
6 PRIMARY KEY (COURSE_ID) );
7 CREATE TABLE DEPT_INFO (
8 DEPT_ID INTEGER NOT NULL,
9 DEPT_NAME VARCHAR(50),
10 PRIMARY KEY (DEPT_ID) );
11 CREATE TABLE OFFICE_INFO (
12 OFFICE_ID INTEGER NOT NULL,
13 OFFICE_NAME VARCHAR(50),
14 HAS_PRINTER SMALLINT,
15 PRIMARY KEY (OFFICE_ID) );
16 CREATE TABLE RACE_INFO (
17 RACE_CODE INTEGER NOT NULL,
18 RACE VARCHAR(50),
19 PRIMARY KEY (RACE_CODE) );
20 CREATE TABLE TRANSCRIPT (
21 USER_ID VARCHAR(50) NOT NULL,
22 COURSE_ID INTEGER NOT NULL,
23 SCORE INTEGER,
24 PRIMARY KEY (USER_ID,
25 COURSE_ID) );
26 CREATE TABLE UNIX_COMMAND (
27 UNIX_COMMAND VARCHAR(50) NOT NULL,
28 CATEGORY VARCHAR(50),
29 PRIMARY KEY (UNIX_COMMAND) );
30 CREATE TABLE USAGE_HISTORY (
31 USER_ID VARCHAR(50) NOT NULL,
32 SESSION_ID INTEGER,
33 LINE_NO INTEGER,
34 COMMAND_SEQ INTEGER,
35 COMMAND VARCHAR(50) );
36 CREATE TABLE USER_INFO (
37 USER_ID VARCHAR(50) NOT NULL,
38 FIRST_NAME VARCHAR(50),
39 LAST_NAME VARCHAR(50),
40 SEX VARCHAR(1),
41 DEPT_ID INTEGER,
42 OFFICE_ID INTEGER,
43 GRADUATE SMALLINT,
```

```

44 RACE INTEGER,
45 PASSWORD VARCHAR(50) NOT NULL,
46 YEARS_USING_UNIX INTEGER,
47 ENROLL_DATE DATE,
48 PRIMARY KEY (USER_ID) );
49 ALTER TABLE COURSE_INFO ADD FOREIGN KEY (OFFERED_DEPT)
50 REFERENCES DEPT_INFO (DEPT_ID);
51 ALTER TABLE TRANSCRIPT ADD FOREIGN KEY (USER_ID)
52 REFERENCES USER_INFO (USER_ID);
53 ALTER TABLE TRANSCRIPT ADD FOREIGN KEY (COURSE_ID)
54 REFERENCES COURSE_INFO (COURSE_ID);
55 ALTER TABLE USAGE_HISTORY ADD FOREIGN KEY (USER_ID)
56 REFERENCES USER_INFO (USER_ID);
57 ALTER TABLE USER_INFO ADD FOREIGN KEY (DEPT_ID)
58 REFERENCES DEPT_INFO (DEPT_ID);
59 ALTER TABLE USER_INFO ADD FOREIGN KEY (OFFICE_ID)
60 REFERENCES OFFICE_INFO (OFFICE_ID);
61 ALTER TABLE USER_INFO ADD FOREIGN KEY (RACE)
62 REFERENCES RACE_INFO (RACE_CODE);

```

B Method courseNameExists

```

1 private static boolean courseNameExists(String courseName) throws Exception {
2     rs = DBUtil.executeQuery("Select *FROM COURSE_INFO
3         WHERE COURSE_NAME ='" + courseName + "'");
4     while (rs.next()) {
5         return true;
6     }
7     //close resultset
8     rs.close();
9     return false;
10 }

```

C Method getCommandsByCategory

```

1 private static List getCommandsByCategory(String category) throws Exception {
2     rs = DBUtil.executeQuery("SELECT UNIX_COMMAND FROM UNIX_COMMAND
3         WHERE CATEGORY = '" + category + "'");
4     List commands = new ArrayList();
5     while (rs.next()) {
6         commands.add(rs.getString("UNIX_COMMAND"));
7     }
8     return commands;
9 }

```

D Method getCourseIDByName

```
1 private static int getCourseIDByName(String courseName) throws Exception {
2   rs = DBUtil.executeQuery("SELECT COURSE_ID FROM COURSE_INFO
3     WHERE COURSE_NAME = '" + courseName + "'");
4   while (rs.next()) {
5     return rs.getInt("COURSE_ID");
6   }
7   //close resultset
8   rs.close();
9   return -1;
10 }
```

E Method getDepartmentIDByName

```
1 private static int getDepartmentIDByName(String name) throws Exception {
2   String sql = "SELECT DEPT_ID FROM DEPT_INFO WHERE DEPT_NAME = '" +
3     name + "'";
4   rs = DBUtil.executeQuery(sql);
5   while (rs.next()) {
6     return rs.getInt("DEPT_ID");
7   }
8   return -1;
9 }
```

B.2.2 RiskIt

A Database Schema

```
1 CREATE TABLE education
2   (SSN int NOT NULL,
3   EDUCATION char(50),
4   EDUENROLL char(50),
5   PRIMARY KEY (SSN) );
6 CREATE TABLE employmentstat
7   (SSN int NOT NULL,
8   UNEMPLOYMENTREASON char(50),
9   EMPLOYMENTSTAT char(50),
10  PRIMARY KEY (SSN) );
11 CREATE TABLE geo
12   (REGION char(50) NOT NULL,
13   RESSTATE char(50) NOT NULL,
14   PRIMARY KEY (RESSTATE) );
15 CREATE TABLE industry
16   (INDUSTRYCODE int NOT NULL,
17   INDUSTRY char(50),
18   STABILITY int,
```

```

19  PRIMARY KEY (INDUSTRYCODE) ) ;
20  CREATE TABLE investment
21  (SSN int NOT NULL,
22  CAPITALGAINS int,
23  CAPITALLOSSES int,
24  STOCKDIVIDENDS int,
25  PRIMARY KEY (SSN) ) ;
26  CREATE TABLE job
27  (SSN int NOT NULL,
28  WORKCLASS char(50),
29  INDUSTRYCODE int,
30  OCCUPATIONCODE int,
31  UNIONMEMBER char(50),
32  EMPLOYERSIZE int,
33  WEEKWAGE int,
34  SELFEMPLOYED smallint,
35  WORKWEEKS int,
36  PRIMARY KEY (SSN) ) ;
37  CREATE TABLE migration
38  (SSN int NOT NULL,
39  MIGRATIONCODE char(50),
40  MIGRATIONDISTANCE char(50),
41  MIGRATIONMOVE char(50),
42  MIGRATIONFROMSUNBELT char(50),
43  PRIMARY KEY (SSN) ) ;
44  CREATE TABLE occupation
45  (OCCUPATIONCODE int NOT NULL,
46  OCCUPATION char(50),
47  STABILITY int,
48  PRIMARY KEY (OCCUPATIONCODE) ) ;
49  CREATE TABLE stateabbv
50  (ABBV char(2) NOT NULL,
51  NAME char(50) NOT NULL ) ;
52  CREATE TABLE userrecord
53  (NAME char(50),
54  ZIP char(5),
55  SSN int NOT NULL,
56  AGE int,
57  SEX char(50),
58  MARITAL char(50),
59  RACE char(50),
60  TAXSTAT char(50),
61  DETAIL char(100),
62  HOUSEHOLDDDETAIL char(100),
63  FATHERORIGIN char(50),
64  MOTHERORIGIN char(50),
65  BIRTHCOUNTRY char(50),
66  CITIZENSHIP char(50),

```

```

67  PRIMARY KEY (SSN) ) ;
68  CREATE TABLE wage
69  (INDUSTRYCODE int NOT NULL,
70  OCCUPATIONCODE int NOT NULL,
71  MEANWEEKWAGE int,
72  PRIMARY KEY (INDUSTRYCODE,
73  OCCUPATIONCODE) ) ;
74  CREATE TABLE youth
75  (SSN int NOT NULL,
76  PARENTS char(50),
77  PRIMARY KEY (SSN) ) ;
78  CREATE TABLE ziptable
79  (ZIP char(5),
80  CITY char(20),
81  STATENAME char(20),
82  COUNTY char(20) ) ;
83  ALTER TABLE education ADD
84  FOREIGN KEY (SSN) REFERENCES userrecord (SSN);
85  ALTER TABLE employmentstat ADD
86  FOREIGN KEY (SSN) REFERENCES userrecord (SSN);
87  ALTER TABLE investment ADD
88  FOREIGN KEY (SSN) REFERENCES userrecord (SSN);
89  ALTER TABLE job ADD
90  FOREIGN KEY (OCCUPATIONCODE)
91  REFERENCES occupation (OCCUPATIONCODE) ;
92  ALTER TABLE job ADD
93  FOREIGN KEY (SSN) REFERENCES userrecord (SSN) ;
94  ALTER TABLE job ADD
95  FOREIGN KEY (INDUSTRYCODE)
96  REFERENCES industry (INDUSTRYCODE);
97  ALTER TABLE migration ADD
98  FOREIGN KEY (SSN) REFERENCES userrecord (SSN);
99  ALTER TABLE wage ADD
100 FOREIGN KEY (INDUSTRYCODE)
101 REFERENCES industry (INDUSTRYCODE) ;
102 ALTER TABLE wage ADD
103 FOREIGN KEY (OCCUPATIONCODE)
104 REFERENCES occupation (OCCUPATIONCODE);
105 ALTER TABLE youth ADD
106 FOREIGN KEY (SSN) REFERENCES userrecord (SSN);

```

B Method createNewUser

```

1  public int createNewUser(String name, String zipcode, long ssn, int age, String sex,
2      String marital, String race, String taxStatus, String taxDetails,
3      String houseHoldDetails, String fatherOrigin, String motherOrigin,
4      String birthCountry, String citizenship , int capitalGains,
5      int capitalLosses, int stockDividend, String parents,

```

```

6         String unEmploymentReason, String employmentStatus,
7         String education, String eduEnroll, String workClass,
8         String unionMember, int industryCode, int occupationCode,
9         int employerSize, int weekWage, int selfEmployed, int workWeeks)
10    {
11    int success = 0, count = 0;
12    Statement stat = null;
13    ResultSet result = null;
14    try {
15    Connection conn = DatabaseConnection.getConnection();
16    stat = conn.createStatement();
17    System.out.println("Customer details you entered are .....");
18    System.out.println(" Name : " + name);
19    System.out.println(" SSN : " + ssn);
20    System.out.println(" Age : " + age);
21    System.out.println(" Sex : " + sex);
22    System.out.println(" Marital Status : " + marital);
23    System.out.println(" Race : " + race);
24    System.out.println(" Tax Status : " + taxStatus);
25    System.out.println(" Tax Details : " + taxDetails);
26    System.out.println(" House hold information : " + houseHoldDetails);
27    System.out.println(" Father's origin : " + fatherOrigin);
28    System.out.println(" Mother's origin : " + motherOrigin);
29    System.out.println(" Birth Country : " + birthCountry);
30    System.out.println(" Citizenship : " + citizenship );
31    System.out.println(" Capital Gains : " + capitalGains);
32    System.out.println(" Capital losses : " + capitalLosses );
33    System.out.println(" Stock dividends : " + stockDividend);
34    System.out.println(" Parents' Informarion : " + parents);
35    System.out.println(" Employment Status : " + employmentStatus);
36    System.out.println(" Un-employment reason : " + unEmploymentReason);
37    System.out.println(" Education : " + education);
38    System.out.println(" Further education enrolled : " + eduEnroll);
39    System.out.println(" Work Class : " + workClass);
40    System.out.println(" Union member : " + unionMember);
41    System.out.println(" Industry code : " + industryCode);
42    System.out.println(" Occupation code : " + occupationCode);
43    System.out.println(" Employer Size : " + employerSize);
44    System.out.println(" Self - Employed : " + selfEmployed);
45    System.out.println(" Week wage : " + weekWage);
46    System.out.println(" No. of weeks of work : " + workWeeks);
47    //Search if the data is already there in table
48    String cmd_dataCheck = "Select *from userrecord where ssn = "
49        + ssn;
50    //get connection to the table and insert all the data
51    String cmd_userRecord = "INSERT INTO userrecord VALUES ('" + name + "', '"
52        + zipcode + "', '" + ssn + "', '" + age + "', '" + sex + "', '"
53        + marital + "', '" + race + "', '" + taxStatus + "', '"

```

```

54         + taxDetails + "'", ' + houseHoldDetails + "'", ' + fatherOrigin
55         + "'", ' + motherOrigin + "'", ' + birthCountry + "'", '
56         + citizenship + '")";
57 String cmd_investment = "INSERT INTO investment VALUES (" + ssn + ", "
58         + capitalGains + ", " + capitalLosses + ", "
59         + stockDividend + '")";
60 String cmd_youth = "INSERT INTO youth VALUES (" + ssn + ", "
61         + parents + '")";
62 String cmd_employment = "INSERT INTO employmentstat VALUES (" + ssn + ", "
63         + unEmploymentReason + "'", ' + employmentStatus + '")";
64 String cmd_education = "INSERT INTO education VALUES (" + ssn + ", "
65         + education + "'", ' + eduEnroll + '")";
66 String cmd_job = "INSERT INTO job VALUES (" + ssn + ", " + workClass + "'", "
67         + industryCode + ", " + occupationCode + ", " + unionMember
68         + "'", " + employerSize + ", " + weekWage + ", " + selfEmployed
69         + ", " + workWeeks + '")";
70 //first check before inserting the data into table
71 result = stat.executeQuery(cmd_dataCheck);
72 while(result.next())
73 {
74     ++ count;
75 }
76 System.out.println("-----");
77 //if no record is present, then insert the data
78 if (count == 0)
79 {
80     stat.execute(cmd_userRecord);
81     stat.execute(cmd_investment);
82     stat.execute(cmd_youth);
83     stat.execute(cmd_employment);
84     stat.execute(cmd_education);
85     stat.execute(cmd_job);
86     success = 1;
87     System.out.println("This customer is successfully added to database");
88 }
89 else
90     System.out.println("This customer is already present in the database");
91
92 result.close();
93 stat.close();
94
95 System.out.println("-----");
96 }
97 catch(Exception e) {
98     System.out.println("Catch in createUser");
99     e.printStackTrace();
100 }
101 return success;

```

C Method deleteUser

```

1  public int deleteUser(String ssn)
2  {
3  int count = 0, success = 0;
4  Statement stat = null;
5  ResultSet result = null;
6  try
7  {
8  String del_user = "Delete from userrecord where ssn = " + Long.parseLong(ssn);
9  String del_job = "Delete from job where ssn = " + Long.parseLong(ssn);
10 String delEdu = "Delete from education where ssn = " + Long.parseLong(ssn);
11 String delEmp = "Delete from employmentstat where ssn = " + Long.parseLong(ssn);
12 String delInv = "Delete from investment where ssn = " + Long.parseLong(ssn);
13 String del_mig = "Delete from migration where ssn = " + Long.parseLong(ssn);
14 String del_youth = "Delete from youth where ssn = " + Long.parseLong(ssn);
15 String cmd_checkData = "Select *from userrecord where ssn = " + Long.parseLong(ssn);
16 Connection conn = DatabaseConnection.getConnection();
17 stat = conn.createStatement();
18
19 result = stat.executeQuery(cmd_checkData);
20 while(result.next())
21     ++ count;
22 //check if records are there and then delete it
23 if (count > 0)
24 {
25     stat.execute(del_youth);
26     stat.execute(del_mig);
27     stat.execute(del_job);
28     stat.execute(delEdu);
29     stat.execute(delEmp);
30     stat.execute(delInv);
31     stat.execute(del_user);
32     success = 1;
33 }
34 //re do the checking to see if the data is deleted
35 result = stat.executeQuery(cmd_checkData);
36 while(result.next())
37     ++ count;
38 if (count > 0)
39 {
40     stat.execute(del_youth);
41     stat.execute(del_mig);
42     stat.execute(del_job);
43     stat.execute(delEdu);
44     stat.execute(delEmp);

```

```
45     stat.execute(delInv);
46     stat.execute(delUser);
47     success = 1;
48 }
49 else
50     System.out.println("The data is already deleted");
51 result.close();
52 }
53 catch(Exception e)
54 {
55     System.out.println("Exception in delete");
56     e.printStackTrace();
57 }
58 return success;
59 }
```

