Verification of Full Functional Correctness for Imperative Linked Data Structures

by

Karen K. Zee

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2010

Author .....
Department of Electrical Engineering and Computer Science
January 29, 2010

Certified by .....
Martin C. Rinard
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chair, Department Committee on Graduate Students

Verification of Full Functional Correctness for Imperative Linked Data Structures

by

Karen K. Zee

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2010, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

We present the verification of full functional correctness for a collection of imperative linked data structures implemented in Java. A key technique that makes this verification possible is a novel, integrated proof language that we have developed within the context of the Jahob program verification system. Our proof language allows us to embed proof commands directly within the program, making it possible to reason about the behavior of the program in its original context. It also allows us to effectively leverage Jahob's integrated reasoning system. Unlike conventional program verification systems that rely on a single monolithic prover, Jahob includes interfaces to a diverse collection of specialized automated reasoning systems—automated theorem provers, decision procedures, and program analyses—that work together to prove the verification conditions that the system automatically generates. Our proof language enables the developer to direct the efforts of these automated reasoning systems to successfully verify properties that the system is unable to verify without guidance.

Our specifications characterize the behavior of the data structures in terms of their abstract state, resulting in verified interfaces that can be used to reason about the behavior of the data structures without revealing the underlying representation. The results demonstrate the effectiveness of our proof language and integrated reasoning approach, and provide valuable insight into the specification and verification of imperative linked data structures.

Thesis Supervisor: Martin C. Rinard
Title: Professor of Computer Science and Engineering

Contents

1 Introduction **15**

1.1 Motivation and Challenges 15
1.2 The Jahob Program Verification System 16
1.2.1 Basic Specification Approach 16
1.2.2 Basic Verification Approach 17
1.2.3 Integrated Reasoning 18
1.2.4 Integrated Proof Language 19
1.2.5 Extending the Standard Verification Framework 21
1.3 Verified Data Structures 21
1.3.1 Experience 22
1.3.2 Empirical Results 23
1.4 Contributions 23
1.5 Summary 24

2 Examples **25**

2.1 Jahob Specifications 25
2.1.1 Concrete State 26
2.1.2 Abstract State 26
2.1.3 Abstraction Function 27
2.1.4 Method Contracts 28
2.1.5 Loop Invariants 28
2.2 Verification 29
2.3 Integrated Proof Language 30
2.3.1 Hash Table Example 31
2.3.2 Identifying Intermediate Lemmas 32
2.3.3 Controlling the Assumption Base 34
2.3.4 Resolving Key Choice Points 35
2.3.5 Decomposing Complex Goals 36
2.3.6 Other Proof Commands 37
2.3.7 Why the Proof Language Works 38
2.3.8 Practical Advantages of the Proof Language 39
2.4 Summary 39

3 The Jahob System **41**
3.1 Jahob Programs . . . 41
3.1.1 Jahob Formulas . . . 42
3.1.2 Specification Variables . . . 44
3.1.3 Method Contracts . . . 45
3.1.4 Class Invariants . . . 45
3.1.5 Annotations Within Method Bodies . . . 46
3.2 Generating Verification Conditions . . . 48
3.2.1 Representation of Program Memory . . . 49
3.2.2 Class-Level Encapsulation . . . 51
3.2.3 From Java to Guarded Commands . . . 51
3.2.4 From Extended to Simple Guarded Commands . . . 54
3.2.5 Translating Extended Guarded Commands . . . 54
3.2.6 Accounting for Variable Dependencies . . . 54
3.3 Proving Verification Conditions . . . 55
3.3.1 Splitting . . . 56
3.3.2 Using Multiple Provers . . . 56
3.3.3 Formula Approximation . . . 58
3.4 Provers Deployed in Jahob . . . 58
3.4.1 Syntactic Prover . . . 58
3.4.2 First-Order Provers . . . 59
3.4.3 SMT Provers . . . 59
3.4.4 MONA . . . 59
3.4.5 BAPA . . . 60
3.4.6 Simple Cardinality Prover . . . 60
3.4.7 Isabelle and Coq . . . 60
3.5 Discussion . . . 62
3.5.1 Specification Language . . . 62
3.5.2 Integrated Reasoning . . . 63
3.5.3 Overall Design . . . 63
3.5.4 Verifying Instantiable Data Structures . . . 64
3.5.5 Limitations . . . 66
3.6 Summary . . . 66

4 The Integrated Proof Language **69**
4.1 The Proof Language Commands . . . 70
4.1.1 The Assumption Base . . . 72
4.1.2 The Note Command . . . 73
4.1.3 The Localize Command . . . 74
4.1.4 First-Order Logic Commands . . . 74
4.1.5 The Induct Command . . . 79
4.2 Soundness . . . 79
4.3 Proof Reuse and Parameterization . . . 80
4.3.1 Verification Feedback . . . 81
4.4 Summary . . . 81

5 Priority Queue **83**
5.1 The Concrete State . . . 83
5.2 The Abstract State . . . 85
5.3 The Abstraction Function and Invariants . . . 85
5.4 Method Contracts for Public Methods . . . 89
5.4.1 The PriorityQueue Constructor . . . 89
5.4.2 The add() Method . . . 89
5.4.3 The clear() Method . . . 90
5.4.4 The peek() Method . . . 90
5.4.5 The poll() Method . . . 90
5.4.6 The size() Method . . . 90
5.5 Method Contracts for Private Methods . . . 91
5.5.1 The parent() Method . . . 91
5.5.2 The left() Method . . . 91
5.5.3 The right() Method . . . 91
5.5.4 The contains() Method . . . 94
5.5.5 The resize() Method . . . 94
5.5.6 The addOnly() Method . . . 95
5.5.7 The heapify() Method . . . 95
5.5.8 The inductProof() Method . . . 96
5.6 Implementation and Verification . . . 96
5.6.1 The peek() Method . . . 96
5.6.2 The inductProof() Method . . . 98
5.7 Discussion . . . 101
5.8 Summary . . . 102

6 Association List **105**
6.1 The Concrete State . . . 105
6.2 The Abstract State . . . 105
6.3 The Abstraction Function and Invariants . . . 107
6.4 Method Contracts for Public Methods . . . 108
6.4.1 The AssociationList Constructor . . . 108
6.4.2 The containsKey() Method . . . 110
6.4.3 The get() Method . . . 110
6.4.4 The isEmpty() Method . . . 110
6.4.5 The put() Method . . . 110
6.4.6 The remove() Method . . . 110
6.4.7 The add() Method . . . 111
6.4.8 The replace() Method . . . 111
6.5 Method Contracts for Private Methods . . . 112
6.5.1 The _containsKey() Method . . . 112
6.5.2 The _add() Method . . . 112
6.5.3 The _remove() Method . . . 113
6.6 Implementation and Verification . . . 113
6.6.1 The get() Method . . . 114

6.6.2 The remove() Method . . . 114
6.6.3 The _containsKey() Method . . . 115
6.6.4 The _remove() Method . . . 115
6.6.5 Ease of Verification versus Efficiency . . . 117
6.6.6 Summary . . . 118

7 Hash Table **121**
7.1 The Concrete State . . . 121
7.2 The Abstract State . . . 121
7.3 Abstraction Function and Invariants . . . 123
7.4 Method Contracts for Public Methods . . . 125
7.4.1 The Hashtable Constructor . . . 125
7.4.2 The containsKey() Method . . . 125
7.4.3 The get() Method . . . 125
7.4.4 The isEmpty() Method . . . 127
7.4.5 The put() Method . . . 127
7.4.6 The remove() Method . . . 127
7.4.7 The add() Method . . . 127
7.4.8 The replace() Method . . . 128
7.5 Method Contracts for Private Methods . . . 130
7.5.1 The compute_hash() Method . . . 130
7.5.2 The _containsKey() Method . . . 130
7.5.3 The bucketContainsKey() Method . . . 130
7.5.4 The _remove() Method . . . 131
7.5.5 The removeFirst() Method . . . 131
7.5.6 The removeFromBucket() Method . . . 132
7.5.7 The _add() Method . . . 132
7.6 Implementation and Verification . . . 132
7.6.1 The get() Method . . . 133
7.6.2 The compute_hash() Method . . . 134
7.6.3 The remove() Method . . . 135
7.6.4 The _containsKey() Method . . . 136
7.6.5 The bucketContainsKey() Method . . . 137
7.6.6 The _remove() Method . . . 138
7.6.7 The removeFirst() Method . . . 139
7.6.8 The removeFromBucket() Method . . . 143
7.7 Discussion . . . 156
7.8 Summary . . . 157

8 Experimental Results **159**
8.1 Verified Data Structures . . . 159
8.1.1 Statistics . . . 160
8.2 Provers . . . 169
8.2.1 Prover Order . . . 170
8.2.2 Sequents Proved . . . 170

8.2.3 Verification Times . . . 173
8.3 Effect of Proof Language Commands . . . 178
8.3.1 Sensitivity to Proof Language Commands . . . 180
8.3.2 Discussion . . . 181
8.4 Summary . . . 181

9 Related Work **183**
9.1 Program Verification . . . 183
9.1.1 Hob . . . 184
9.1.2 Jahob . . . 184
9.1.3 ESC/Modula-3 and ESC/Java . . . 185
9.1.4 ESC/Java2 . . . 185
9.1.5 Spec# . . . 186
9.1.6 KeY . . . 186
9.1.7 Krakatoa . . . 187
9.1.8 LOOP . . . 187
9.1.9 Jive . . . 188
9.1.10 Jack . . . 188
9.2 Interactive Theorem Provers . . . 188
9.2.1 Isabelle . . . 189
9.2.2 Coq . . . 190
9.2.3 ACL2 . . . 191
9.2.4 KIV . . . 192
9.3 Data Structure Verification . . . 192
9.3.1 Shape Analysis . . . 192
9.3.2 Separation Logic . . . 194
9.3.3 Type Systems . . . 196
9.3.4 Decidable Logics . . . 196
9.4 Finitization and Automated Testing . . . 197
9.5 Summary . . . 197

10 Conclusion **199**
10.1 Future Directions . . . 200
10.2 Summary . . . 201

A Soundness Proofs **203**

Acknowledgments

(Computer Science: A Love Story)

Once upon a time, there was a girl who loved computer science, who lived in the Promised Land of California. There, she had family and friends, and was very happy. Now, far, far away from the Promised Land was a magical kingdom known as Emeyetee. In this kingdom, there was a small village called Elsee Ess, where lived many of the most brilliant computer scientists in the world. Boys and girls from all over the world came to the village of Elsee Ess, where they became apprentices to master computer scientists. There they were trained in the art of computer science, and learned many things, in the hopes of one day becoming master computer scientists themselves.

Now, to become a master computer scientist in the kingdom of Emeyetee, an apprentice had to embark on a long and dangerous quest to capture and bring back a mystical creature known as the "thesis." This was understood to be a very difficult task, and one that would take many years. But the girl who loved computer science loved it a lot, and she longed to become a master computer scientist. So one fine day, she left her family and friends in the Promised Land, and traveled all the way to the village of Elsee Ess in the kingdom of Emeyetee.

When she arrived at Elsee Ess, she came across a master computer scientist called Martin, one of the most brilliant computer scientists in the land. She told him about her quest, and because he was not only brilliant, but also very kind, he agreed to accept her as an apprentice and to help her find a thesis of her very own.[1] Martin had many apprentices, and their names were: Radu, Maria-Cristina, Darko, C. Scott, Brian, Alex, Viktor, Patrick, Chandra, Wes, Amy Williams, Michael Carbin, Zoran, Sasa, Deokhwan, Kostas, Vijay, Stelios, Jeff, Suhabe, Bruno, Charles, Hai, Philippe, Lionel, Jérémie, and Yungbum. Although they were all on their own quests, they helped one another, and the girl learned many things from them.[2] She and Viktor, along with Thomas, Charles, Hai, Lionel, Jérémie, and Andreas (computer scientists from far away places), developed an ingenious contraption for capturing the elusive thesis, which they named Jahob.[3] They were aided by Martin, and by a council known

[1]Copious thanks go to my advisor, Martin Rinard, for his unwavering enthusiasm, insightful guidance, and steadfast support in matters both technical and mundane. I am grateful for his many contributions to my research and to the papers we co-authored. He has been a generous donor of dark chocolate in critical moments. I will also miss his delectable culinary creations.

[2]Many thanks go to the past and present members of Martin's research group, for their friendship, support, and many interesting discussions. Particular thanks go to C. Scott Ananian and Brian Demsky, for their help getting started with Flex, way back when; to Alexandru Sălcianu, for his superior knowledge of abstract interpretation; to Kostas Arkoudas, for introducing me to the Athena proof system and for his work on the paper we co-authored; to Patrick Lam, for working with me on Hob, and for his expertise in all things Linux; to Viktor Kuncak, for working with me through many Samoan midnight deadlines on Hob and Jahob papers; and to Michael Carbin and Sasa Misailovic, for their proofreading prowess.

[3]The work presented in this thesis would not have been possible without the many colleagues who worked with me on the Jahob system, and the anonymous reviewers of our submitted papers, who helped us to retool and refine our ideas. Many thanks go to Viktor Kuncak, for starting the Jahob project, for his many contributions to the system and to the papers we co-authored,

as the "anonymous reviewers," who helped them make sure that their contraption was everything that a contraption should be. She was also guided by other master computer scientists, whose names were Saman, Daniel Jackson, Krste, Srini, Anant, Berthold, and Barbara;[4] and helped by many other villagers who lived in Elsee Ess, whose names were Allen, Eugene, Dave, Jean, Greg, Michal, Derek, Jonathan, Mary McDavitt, Maria, Marilyn, Janet, Anne, Cornelia, Michael Vezza, and Shireen, to name only a few.[5] In time, the village of Elsee Ess joined forces with a nearby village to become the greater village of Seasail, where the girl continued to study.

Now, there were also many wonderful people who lived in the kingdom of Emeyetee, and the girl met and became friends with many of them. In fact, she became friends with so many of them that there is not room here to list all of them, or all of the adventures that they had together. But the ones with whom she had the greatest adventures were Michael Taylor, Angelina, Amy Williams, Shiyun, Livia, Karen Shu, Amy Hung, Jen, Anna, Wouter, Eileen, Po-Ru, Ji-Jon, Felicity, Ben, Rachel, Emily, Mobolaji, Marie-Eve, Sungyon, Laura, Yushi, John, Angie, Albert Chow, Barney, Albert Huang, Tsoline, and Vinson.[6] Among her friends were also many who belonged to the select bands of warriors known as Geecee Eff, Kross Linked, and Kross

---

and for introducing the proof hints that led to the development of the integrated proof language. Thomas Wies worked extensively on both the shape analysis component Bohne and on the Jahob infrastructure at large. I am thankful to him and his advisor Andreas Podelski for their work on the papers we co-authored. I also thank the many visiting students who worked on Jahob. Special thanks go to Charles Bouillaguet, who developed the Jahob interface for first-order provers; Huu Hai Nguyen, who implemented the BAPA decision procedure for Jahob; Lionel Rieg, who worked on commutativity analysis; and Jérémie Dimino, who worked on translating temporal logic specifications into Jahob specifications.

[4]I thank my committee members Daniel Jackson and Saman Amarasinghe, who provided indispensable guidance and feedback. Their recommendations led to many improvements in this thesis. Special thanks go to Daniel for his insightful questions, which helped me to elucidate the novel aspects of the system; and to Saman, for taking me under his wing and adopting me into his research group at conferences where I was flying solo. I am grateful to Krste Asanović and Srini Devadas for their tutelage during my 6.823 teaching assistantship. I also thank my academic advisors Anant Agarwal, Srini Devadas, and Berthold Horn for their patience and guidance throughout my graduate career. I thank Barbara Liskov, who taught me 6.170 in undergrad, and gave me the courage to apply to the MIT graduate program in the first place.

[5]I have benefited greatly from my interactions with the students, faculty, researchers, and staff at CSAIL and in the EECS department here at MIT. As first years, Allen Miu, Eugene Shih, and I spent many happy hours hacking on autonomous LEGO vehicles in Eugene's lab. I am grateful to the past and current denizens of G7 (and, prior to that, the 6th floor of NE43) for making lab a welcoming place. Particular thanks go to David Wentzlaff, Jean Yang, Greg Little, Michal Karczmarek, Derek Rayside, and Jonathan Babb, for many interesting conversations. I thank Mary McDavitt, who takes care of all of us on the floor, and went to extraordinary lengths to get me the most amazing thesis defense cake ever. I also thank Maria Rebelo, Marilyn Pierce, Janet Fischer, Anne Hunter, Cornelia Colyer, Michael Vezza, and Shireen Agah, for tactical support and encouragement.

[6]Special thanks go to Michael, for initiating me into the secrets of the MBTI, for his companionship during my Taylorian travels, and for all that we learned together about life, the universe, and tiled architectures; to Angelina, Amy Williams, and Shiyun, whose friendship has blessed me in so many ways over the years; to Livia, for her wisdom, humor, prayers, and the definitive ISTJ perspective; to my book club, for their friendship and faithfulness; and to the past and present members of my bible study, who have all inspired me and supported me in different ways.

Products.[7] She enjoyed her adventures with all of them, and was grateful for their friendship and kindness.

Although the girl spent many years sojourning in the kingdom of Emeyetee, she was not forgotten by her family and friends in other parts of the world. These include her father and mother, Queenie, Jasper, Connie, Amelia, Raymond, Yuna, Jack, Vikki, Anca, Victor, Nathaniel, and the Wendel family.[8] She was also helped by many wise and loving people at Emeyetee, who watched over her and counseled her on her quest. Their names were Dawn, Lisa, Bina, Ann and Terry Orlando, Lori Lerman, Denise Lanfranchi, Ann McLaughlin, Mary Thompson, and Kevin Ford.[9]

Then, one day, after many years of adventures in the land of Emeyetee, she was wandering across a clearing, with the ingenious contraption in hand, when she came across the elusive thesis. After an epic battle, she captured it and brought it back to her brilliant advisor Martin. He and his fellow master computer scientists Daniel and Saman authenticated the ferocious beast, and declared her apprenticeship completed.

Not only that, but during her many adventures, she met a wonderful boy named Daniel Wendel, who was both loving and kind. He helped her in her quest in many ways, and inspired her with his patience, love, and faith. After she completed her apprenticeship, they continued to have adventures together, and to the best of my knowledge, they lived happily ever after.

The End

> *Trust in the Lord with all your heart and lean not on your own understanding; in all your ways acknowledge him, and he will make your paths straight.* Proverbs 3:5–6

[7]I am grateful to my friends in GCF, Cross Linked, and Cross Products, who have greatly enriched my life with their friendship, and inspired me with their great love for God's kingdom.

[8]My family and friends have been a constant source of strength and encouragement throughout my graduate career. I thank my dad, my mom, Bean, Jasper, Aunt Connie, Aunt Amelia, Raymond, and Yuna for their love and support. I thank Jack, who first inspired me to take 6.001, and whose friendship and encouragement have sustained me throughout the years. I thank Vikki Rubens, Anca Brad, and Victor Luchangco for their wisdom and prayers. I thank Nathaniel, for many late night phone conversations. I thank the Wendel family for their warm welcome and for providing me with a home away from home.

[9]I am grateful to my many wise counselors for their guidance, encouragement, and prayers.

[10]Finally, I thank Daniel J. Wendel, whose patience, love, faith, and companionship continues to sustain and inspire me. I am grateful for all the adventures we've had together, and all the adventures yet to come.

Chapter 1

Introduction

This thesis presents techniques and correctness results for the verification of *full functional correctness* for a collection of imperative linked data structure implementations. Specifically, we verify these data structure implementations with respect to formal specifications that completely capture all aspects of the data structure behavior that are relevant to a client of that data structure (with the exception of properties involving execution time and/or memory consumption). Our specifications characterize the behavior of the data structures in terms of their abstract state, resulting in verified interfaces that can be used to reason about the data structure without revealing the underlying data representation.

A key technique that makes this verification possible is a novel, integrated proof language that we have developed in the context of the Jahob program verification system. This language allows the developer to direct complex verifications at the program level, unlike other program verification systems, where developer guidance occurs at the level of failed verification condition formulas. The integrated proof language enables Jahob to provide a unified interface to the developer for the specification and verification process. The proof language is also able to effectively leverage Jahob's integrated reasoning system. Where most conventional program verification systems support a single monolithic prover [51, 56, 110, 14], Jahob includes interfaces to a diverse collection of internal and external automated reasoning systems—automated theorem provers, decision procedures, and program analyses—which work together to prove the verification conditions that the system automatically generates. The proof language is also able to provide additional coordination to the integrated reasoning system, enabling multiple provers to work together to solve a single proof task. Together, these two techniques enable us to use Jahob to verify complex program correctness properties that the system would not otherwise be able to verify.

1.1 Motivation and Challenges

Data structure implementations lend themselves naturally to full functional verification, not only because they are generally small, making the proof task tractable, but also because their interfaces are well understood and can be concisely defined for-

mally. The pervasiveness of linked data structures (which include lists, trees, graphs, and hash tables) in modern software systems also speaks to the practical relevance of such verification. In particular, the prevalence of data structure libraries increases the advantage of verifying data structure implementations, as the costs of verification can be amortized over many uses.

At the same time, imperative linked data structures, the focus of this thesis, present special challenges. Due to the phenomena of aliasing and indirection, many of the desired correctness properties involve logical constructs such as transitive closure and quantifiers that are known to be intractable for automated reasoning systems [70, 86]. In the past, researchers have worked around this problem in one of two ways. The first is to focus on more tractable goals—verify some (but not all) of the desired correctness properties [91, 83, 160, 64, 95, 144, 38, 89, 166, 7], work with programs that do not manipulate recursive linked data structures [153], or use finitization to check correctness properties within a bounded analysis scope [141, 48]. Another approach is to embrace the necessary developer effort by focusing on interactive tools that enable the verification of arbitrarily difficult programs and properties. Research in this area has led to the development of a diverse collection of interactive theorem provers, including the popular Isabelle/HOL [122] and Coq [24] proof assistants.

1.2 The Jahob Program Verification System

In this thesis, we adopt a hybrid approach that takes advantage of automated reasoning techniques but also enables developers to direct the efforts of automated reasoning systems where necessary to resolve difficult proof tasks. We have developed Jahob, a program verification system that uses the standard specification and verification paradigm, but distinguishes itself from other program verification systems in two main ways. First, instead of using a single monolithic prover, Jahob includes interfaces to a diverse collection of automated provers, decision procedures, and program analyses, which work together to prove the verification conditions that Jahob automatically generates. Second, Jahob incorporates a declarative proof language that enables the developer to guide the efforts of the automated reasoning systems to successfully verify properties that the systems are unable to verify without guidance.

1.2.1 Basic Specification Approach

Verification in Jahob starts with the program (or program component, such as a data structure) and the specifications that capture the properties we would like to prove about it. Jahob verifies programs written in a subset of Java. Our specifications use abstract sets and relations to characterize the abstract state of the program. A verified abstraction function establishes the correspondence between the concrete values that the implementation manipulates when it executes and the abstract sets and relations in the specification. Method preconditions and postconditions written in classical higher-order logic use these abstract sets and relations to express externally observable properties of the program.

Classical higher-order logic is particularly effective for specifying data structure implementations because it naturally supports a number of constructs:

- quantifiers for invariants in programs that manipulate an unbounded number of objects,
- a notation for sets and relations, which we use to concisely specify data structure interfaces,
- transitive closure, which is essential for specifying important properties of recursive data structures,
- the cardinality operator, which is suitable for specifying numerical properties of data structures, and
- lambda abstraction, which can represent definitions of per-object specification fields and is useful for parameterized shorthands.

By expressing specifications in terms of sets and relations, developers can soundly hide data structure implementation details and provide intuitive method interfaces. Data structure clients can then use these interfaces to check that the data structure is used correctly and to reason about the effect of data structure operations.

1.2.2 Basic Verification Approach

Jahob proves the desired correctness properties for a program by first generating verification condition formulas, then proving these formulas. The verification conditions are proof obligations that, together, ensure that the program respects method preconditions, postconditions, invariants, and preconditions of operations such as array accesses and pointer dereferences. Jahob's verification condition generator requires loop invariants, which are typically supplied by the developer, but may also be automatically generated using Bohne [160], an algorithm for inferring loop invariants using symbolic shape analysis implemented in Jahob.

The verification condition formulas generated by Jahob are expressed in an undecidable fragment of higher-order logic, and are therefore beyond the reach of any automated decision procedure. Simple attempts to improve the tractability by limiting the expressive power of the logic fail because some of the correctness properties involve inherently intractable constructs such as quantifiers, transitive closure, and lambda abstraction. But although the verification conditions as a whole can be quite complex, they can also be represented as a conjunction of a large number of smaller subformulas, many of which are straightforward to prove. The remaining subformulas, while containing a diverse group of powerful logical constructs, often have enough structure to enable the successful application of specialized decision procedures or theorem provers. Specifically, some subformulas can be proved with sufficient quantifier instantiations, congruence closure algorithms, and linear arithmetic solvers; precise reasoning about reachability is sufficient to discharge others; still others require complex quantifier reasoning but do not require arithmetic reasoning. Armed with this

insight, we developed an *integrated reasoning* approach that enables the simultaneous application of a diverse group of interoperating automated reasoning systems to prove each verification condition.

1.2.3 Integrated Reasoning

Our integrated reasoning approach is based on the following techniques:

- **Splitting:** Jahob splits verification conditions into equivalent conjunctions of subformulas and processes each subformula independently. It can therefore use different provers to establish different parts of proof obligations. Because it treats each prover as a black box, it is easy to incorporate new provers into the system. Moreover, each prover can run on a separate processor core, reducing the running time on modern workstations.

- **Formula Approximation:** Advances in automated theorem proving, decision procedures, and program analysis have produced a diversity of automated reasoning tools that are extremely effective on specialized problems. Jahob uses a variety of new and existing internal and external decision procedures, program analyses, SMT provers, and first-order theorem provers, each with its own restrictions on the set of formulas that it will accept as input. Several formula approximation techniques make it possible to successfully deploy this diverse set of reasoning systems together within a single unified reasoning framework. These approximation techniques accept higher-order logic formulas and create equivalent or semantically stronger formulas accepted by the specialized decision procedures and provers.

  Our approximation techniques rewrite equalities over complex types such as functions, apply beta reduction, and express set operations using first-order quantification. They also soundly approximate constructs not directly supported by a given specialized reasoning system, typically by replacing problematic constructs with logically stronger and simpler approximations.

  Decision procedures such as MONA [67] perform reasoning under the assumption that the models of given formulas are trees. The Jahob interfaces to such decision procedures recognize subformulas that express the relevant structure (such as treeness or transitive closure). They then expose this structure to the decision procedure by applying techniques such as field constraint analysis [159] and encoding transitive closure using second-order quantifiers.

Together, these techniques make it possible to productively apply arbitrary collections of specialized reasoning systems to complex higher-order logic formulas. Our implemented system contains a simple syntactic prover, a simple cardinality prover, interfaces to first-order provers (SPASS [156] and E [149]), an interface to SMT provers (CVC3 [60] and Z3 [116]), an interface to MONA [132], and an interface to the BAPA decision procedure [85, 87].

In practice, the syntactic prover quickly disposes of many of the conjuncts in each verification condition. A complex core of subformulas makes it through to the more

powerful automated reasoning systems. Each of these reasoning systems proves the subset of subformulas for which it is applicable; together, they prove the majority of the remaining conjuncts. When the automation does not succeed (typically due to conjuncts that contain large numbers of universally quantified assumptions), we manually guide the proof process using Jahob's integrated proof language.

1.2.4 Integrated Proof Language

Jahob's integrated proof language enables users to guide the automated reasoning systems in proving verification conditions that the provers are otherwise unable to verify. Many program verification systems, including Jahob, include interfaces to external interactive theorem provers, or proof assistants. In theory, developers can use these interfaces to interactively prove difficult lemmas, making it possible to solve program verification problems requiring arbitrarily complex reasoning. In practice, the complexity of the verification conditions that the system generates and the difficulty of using the proof assistant—an external system with very different basic concepts, capabilities, and limitations than that of the program verification system—hampers the developer's ability to perform these proofs manually. Using a separate development environment for proofs also divorces the proofs from their original context within the annotated program. This approach limits developers to the tools supported by the proof assistant, and denies them access to the substantial automated reasoning power available via the Jahob prover interfaces.

In contrast, Jahob's integrated proof language enables developers to control proofs of program correctness properties while remaining completely within a single unified programming and verification environment. The proof commands are directly included in the annotated program, in a natural extension of the standard assertion mechanism present in many programming languages. Because the proof commands occur at the level of the program, the formulas involved are much easier to understand and verify. Common sequences of proof commands can also be parameterized and reused using standard Java methods.

The proof commands produce verification conditions which are then verified by the underlying system as part of the standard program verification workflow. Because the proof language is seamlessly integrated into the verification system, all of the automated reasoning capabilities of the Jahob system are directly available to the developer. We have found that this availability enables developers to avoid the use of external interactive theorem provers altogether. Instead, developers simply use the Jahob proof language to resolve key choice points in the proof search space. Once these choice points have been resolved, the automated provers can then perform all of the remaining steps required to discharge the verification conditions. This approach effectively leverages the complementary strengths of the developer and the automated reasoning system by allowing the developer to communicate key proof structuring insights to the reasoning system. These insights then enable the reasoning system to successfully traverse the (in practice large and complex) proof search space to obtain formal proofs of the desired verification conditions. In particular, our proof language enables the following techniques, which we have found effective in data structure

verification:

- **Lemma Identification:** The developer can identify key lemmas for the Jahob reasoning system to prove. These lemmas can then help the reasoning system find an appropriate proof decomposition. Such a proof decomposition can be especially important when multiple provers must cooperate to prove a single correctness property. In this case separating the property into lemmas, each of which contains facts suitable for a specific prover, then combining the lemmas, may be the only way to obtain a proof.

- **Witness Identification:** The developer can identify the witness that enables the proof of an existentially quantified verification condition. Because there are, in general, an unbounded number of potential witnesses (very few of which may lead to a successful proof), the difficulty of finding an appropriate witness is often a key obstacle that prevents a fully automated system from obtaining a proof. Our results show that enabling the developer to remove this key obstacle usually leaves the automated system easily able to successfully navigate the proof search space to prove the desired correctness property.

- **Quantifier Instantiation:** The developer can identify how to instantiate specific universally quantified formulas. The potentially unbounded number of possible quantifier instantiations can make developer insight particularly useful in enabling successful proofs.

- **Case Split Identification:** The developer can identify the specific cases to analyze for case analysis proofs.

- **Induction:** The developer can identify an induction variable and induction property that lead to a successful proof by induction. This technique is particularly useful since current theorem provers are not able to automatically prove lemmas requiring induction.

- **Assumption Base Control:** Modern theorem provers are usually given a set of facts (we call this set the *assumption base*), then asked to prove a consequent fact that follows from this set. An assumption base that contains irrelevant facts can produce an overly large proof search space that impedes the ability of the provers to find a proof of the consequent. Our integrated proof language enables developers to control the assumption base (and thereby productively focus the proof search space on the property of interest) by identifying a set of relevant facts for the provers to use when proving a specific verification condition. We have found this functionality essential in enabling modern provers to successfully prove the complex verification conditions that arise in proofs of sophisticated program correctness properties.

Jahob's proof language complements integrated reasoning by leveraging developer insight to direct the provers in discharging the lemmas necessary to the overall proof task. Where formula splitting decomposes proof obligations syntactically, the proof

language enables developers to semantically decompose complex proof obligations that might otherwise be beyond the reach of any single specialized automated reasoning system.

1.2.5 Extending the Standard Verification Framework

The implementation of the integrated proof language soundly translates the proof language commands into the guarded command language (from which the system generates verification conditions). The proof commands translate, in effect, to combinations of assert and assume commands, which encode the proof strategy specified by the proof language commands. Unlike other verification systems, whose verification conditions specify only what formulas need to be proved to demonstrate the correctness of the program, Jahob extends the standard verification approach to specify not only what formulas need to be proved, but also how to prove them. In effect, Jahob generalizes the standard verification condition generation approach to produce not only proof obligations but also proof strategies for the automated provers integrated into the system.

We prove the soundness of our translation mechanism using the weakest liberal preconditions algorithm that Jahob uses to generate verification conditions. Chapter 4 (Section 4.2) and Appendix A present the proofs, which proceed by structural induction on the proof language commands.

1.3 Verified Data Structures

We have implemented our integrated reasoning approach and proof language within the Jahob program verification system, and used the system to specify and verify the full functional correctness of a collection of imperative linked data structures implementations. The verified data structures include both recursive data structures such as lists and trees, as well as array-based data structures such as binary heaps and hash tables. All of the verified data structures were designed to capture the core functionality of the data structure, including standard but hard-to-verify operations such as add and remove, and data structure specific operations such as removing the maximal element from a priority queue. The array list, hash table, and priority queue data structures were modeled after the corresponding implementation and/or interface in java.util, typically modified to account for features not supported by Jahob, such as exceptions and dynamic dispatch. For each data structure, we verified method interfaces that capture all the properties relevant to the data structure client, as well as invariants and an abstraction function that together ensure the correctness of the implementation. The verified data structures are as follows:

- **Array List:** A list stored in an array implementing a map from integers to objects, optimized for storing maps from a dense subset of the integers starting at 0 (modeled after java.util.ArrayList). Method contracts in the list describe operations using an abstract relation $\{(0, v_0), \ldots, (k, v_k)\}$, where $k + 1$ is the number of stored elements.

- **Association List:** A singly-linked list data structure that implements a map interface. Properties verified include the injectivity of both the list contents (each key maps to no more than one value in the map) and the nodes that form the underlying structure of the list (a node may be the first node in a list or else be pointed to by the next field of at most one other node).

- **Binary Search Tree:** A binary search tree implementing a set, with tree operations verified to preserve tree shape, ordering, and changes to tree contents.

- **Circular List:** A circular doubly-linked list implementing a set interface.

- **Cursor List:** A list with a cursor that can be used to iterate over the elements in the list and, optionally, remove elements during the iteration. Method contracts include changes to the list contents and to the position of the iterator. Verified properties include that each complete pass over the list visits each element exactly once.

- **Hash Table:** A hash table that implements a map interface modeled after java.util.Hashtable. The implementation uses separate chaining to resolve conflicts. Verified properties include that each key in the hash table is stored in the correct bucket.

- **Priority Queue:** A priority queue data structure, implemented using a binary heap stored in an array. Its interface is modeled after java.util.PriorityQueue. Verified properties include the binary heap property (an element stored at a given node has a priority greater than or equal to the elements stored at the children of that node) and the resulting heap-global property (the root node contains an element with the maximal priority).

- **Singly-Linked List:** A null-terminated singly-linked list implementing a set interface.

1.3.1 Experience

Our experience shows that while simple data structures, such as the singly-linked list and association list, verify automatically, more complex data structures, such as the priority queue and hash table, require substantial developer guidance in the form of proof commands to fully verify. We also found that destructive updates in the form of methods that directly modify the contents of the data structure are the most difficult to verify, making up almost the entire verification effort. While it is not surprising that observer methods (methods that do not modify state) were among the easiest to verify, we found that methods that indirectly modify the contents of the data structure (i.e. by calling other methods) also required little or no verification effort. This result implies that data structure clients that do not directly manipulate the pointer structure of the program may be good targets for automatic verification.

1.3.2 Empirical Results

The results from our successful verification of these data structures also support the use of our hybrid approach. We used seven different provers to verify the data structure implementations, including the first-order prover SPASS, SMT provers Z3 and CVC3, and the decision procedure MONA. All of the data structures required the use of multiple provers to verify, supporting the use of a diversity of automated reasoning techniques. Even for the most difficult data structures, the large majority of the lemmas that needed to be proved were discharged automatically. However, a small fraction remained that required nontrivial developer guidance in the form of proof commands. In these cases, Jahob's proof language was effective in providing the necessary guidance to verify properties that were otherwise beyond the reach of the automated techniques. We were able to fully verify all the data structure implementations by incorporating proof commands in Jahob's proof language, without relying on proofs written using external proof assistants.

Our results also show that, although the large majority of the data structures require some developer guidance in the form of proof commands, two of the data structures, including an association list implementing a map interface, verified without any guidance, highlighting both the effectiveness of the integrated reasoning approach in automatically verifying complex correctness properties, as well as the utility of our proof language for guiding the verification of the majority of our data structures. The resulting verified interfaces capture all the relevant properties of the data structures from the client perspective, enabling client programs to reason about the behavior of the data structure without probing into the internal details of the implementation.

1.4 Contributions

This thesis makes the following contributions:

- **Verified Data Structures:** It presents a verified collection of imperative linked data structure implementations. To the best of our knowledge, this is the first verification of full functional correctness for a collection of standard imperative linked data structures including lists, trees, and hash tables.

- **Integrated Proof Language:** It presents a declarative proof language for the Jahob program verification system, a translation of the proof language commands into a simple guarded command language, and proofs that demonstrate the soundness of our translation. Note that this approach generalizes the standard specification and verification approach by providing a mechanism by which developers can guide, directly within the annotated program, the proof of difficult properties.

- **Observations and Empirical Results:** It presents our experience using the Jahob program verification system in the above verification, and empirical results characterizing the verification process. Our results show that our

integrated reasoning approach and integrated proof language were effective in verifying our data structure implementations—the large majority of our data structures required multiple provers to verify, indicating that a diverse collection of specialized reasoning systems may be more effective than any single prover, while our integrated proof language enabled us to verify complex properties of data structures that the system could not verify without guidance. Our results also show that programs that indirectly manipulate data structures, through called methods, are substantially easier to verify than data structure implementations, suggesting that data structure clients may be promising candidates for automatic verification.

1.5 Summary

In this thesis, we describe the Jahob program verification system and our experience using this system to verify a collection of imperative linked data structure implementations. In contrast to fully-automated techniques that verify only partial correctness properties, our goal is full functional verification. Jahob implements a hybrid approach that specifically targets this goal, using integrated reasoning to incorporate a diversity of automated reasoning systems and an integrated proof language to enable developers to direct the efforts of the combined reasoning system in difficult proof tasks. Our results show that these two techniques are complementary. By enabling the combined application of a diverse collection of specialized automated provers, decision procedures, and program analyses on the generated proof obligations, the integrated reasoning system is able to prove the large majority of the generated lemmas automatically. We were then able to use our integrated proof language to guide the system in proving the remaining lemmas. The integrated nature of the proof language made it possible to provide only the necessary guidance to the system, which then leveraged the strength of the automated reasoning systems in performing the resulting proof tasks. The result is a verified collection of imperative linked data structure implementations that capture the behavior of the data structures without revealing the underlying data representation.

The following chapters illustrate the use of the Jahob program verification system using examples taken from verified data structures (Chapter 2), and describe the Jahob verification system (Chapter 3) and integrated proof language (Chapter 4). They also describe in detail the verification of three of the above data structure implementations: the priority queue (Chapter 5), the association list (Chapter 6), and the hash table (Chapter 7). Chapter 8 shows the experimental results from the verification. Chapter 9 surveys the related work. Chapter 10 concludes, and draws from our experience to identify promising directions for future work. Appendix A contains the correctness proofs for the proof language command implementation, which is a translation from extended guarded command language to simple guarded command language.

Chapter 2

Examples

This chapter illustrates the use of Jahob specification constructs and proof language commands using excerpts from verified data structures. We use an association list data structure to illustrate the use of the specification constructs, and a hash table data structure to illustrate the use of the proof language commands.

Jahob operates on programs written in a subset of Java and annotated with specifications, loop invariants, and proof language commands. These annotations are written as special Java comments of the form /*: ... */ and //: ..., enabling the use of standard Java tools and compilers on annotated programs. The annotations describe the abstract and concrete state of the program using formulas in higher-order logic. From the annotated program, Jahob generates verification condition formulas. These formulas are then proved by the automated reasoning systems integrated into Jahob, enabling the successful verification of our data structure implementations.

2.1 Jahob Specifications

Figure 2-1 presents an excerpt from an association list data structure implementation annotated with Jahob specifications. The association list is a singly-linked list that implements a map interface. Chapter 6 describes this data structure in more detail, but here we use it to illustrate some common Jahob specification constructs.

Jahob specifications capture the desired behavior of the program using a fairly standard specification approach. The main specification constructs are specification variables, class invariants, and method contracts. Specification variables are variables that Jahob uses in the specification and verification, but which do not exist in the program at run-time. They capture the abstract state of the program. Class invariants specify additional constraints on the abstract and concrete state that the program should preserve. Method contracts describe the behavior of the methods in terms of the abstract state.[1]

Most data structures have simple interfaces, but complex implementations. Jahob allows developers to use the abstract state to specify the behavior of the program

[1] In some cases, method contracts may also refer to the concrete state of the program. For details, see Chapter 3, Section 3.1.3.

```
public /*: claimedby AssociationList */ class Node {
    public Object key;
    public Object value;
    public Node next;
    //: public ghost specvar con :: "(obj * obj) set" = "∅"
}

public class AssociationList {
    private Node first;
    /*:
      public specvar contents :: "(obj * obj) set";
      vardefs "contents == first..con";

      invariant ConDef: "∀x. x ∈ Node ∧ x ∈ alloc ∧ x ≠ null →
        x..con = {(x..key, x..value)} ∪ x..next..con ∧
        (∀v. (x..key, v) ∉ x..next..con)";
      invariant ConNull: "null..con = ∅";
      ...
      invariant MapInv:
      "∀k v0 v1. (k, v0) ∈ contents ∧ (k, v1) ∈ contents → v0 = v1";
      ...
    */
    ...
}
```

Figure 2-1: AssociationList Example

in terms of its interface without revealing the (potentially complex) underlying data representation. Given method contracts that describe the program behavior using the abstract state, data structure clients can depend on these contracts without being affected by implementation changes that preserve the original contract.

2.1.1 Concrete State

Figure 2-2 illustrates the concrete and abstract state of the association list data structure presented in Figure 2-1. The concrete state consists of the Java fields of the Node and AssociationList classes. Each Node object contains key, value, and next fields. The key and value fields hold the key-value pair stored at the given Node object, while the next field refers to the next node in the linked list. The concrete state of an AssociationList object consists of the first field, which stores the first Node object in the linked list. The claimedby annotation in the declaration of the Node class (in Figure 2-1) indicates that its fields may only be modified by methods in the AssociationList class.

2.1.2 Abstract State

The abstract state of the association list consists of the specification variables of the Node and AssociationList classes. The abstract state of each Node object consists of

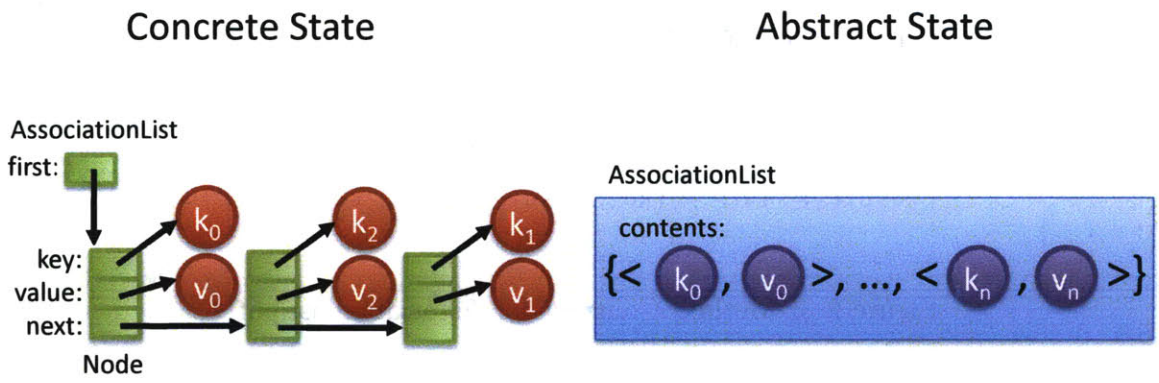


Figure 2-2: Concrete and Abstract State of AssociationList

the specification variable con. It represents the mappings stored in the linked list starting at the given node as a set of key-value pairs.

The abstract state of each AssociationList object consists of the specification variable contents. It represents the mappings stored in the association list as a set of key-value pairs. The value of this variable is defined in terms of concrete Java variables and/or other specification variables.

2.1.3 Abstraction Function

While the program code determines the actual behavior of the program by operating on its concrete state, the specification describes the desired behavior using the abstract state. To verify that the program conforms to its specification, Jahob therefore needs to know the relationship between the program's concrete and abstract state. This relationship is given by the abstraction function.

In the AssociationList class, the vardefs declaration for the specification variable contents gives the definition for contents. It defines contents as the value of con for the first node in the linked list. The ".." notation used in Jahob formulas for field dereference is equivalent to the Java "." notation.

The ConDef and ConNull invariants give the relationship between the specification variable con and the concrete state of the program. The ConDef invariant defines the value of con for a given node as the union of the singleton set containing the key-value pair at the given node, and the value of con for the next node in the list. It additionally requires that the key at the given node not be mapped to a value in the rest of the list. The ConNull invariant specifies that the value of con for null is the empty set. Together, the ConDef and ConNull invariants illustrate the use of invariants for expressing reachability using recursion. The ConNull invariant gives the base case. The ConDef invariant gives the recursive case.

The association list data structure contains other invariants that specify constraints on the abstract and concrete state. For example, the MapInv invariant requires that each key in the association list map to a unique value. Invariants express important constraints that the data structure must preserve for its correct operation. Some invariants express the lack of sharing between the concrete state of different

```
public Object put(Object k0, Object v0)
/*: requires "k0 ≠ null ∧ v0 ≠ null"
    modifies contents
    ensures "contents = old contents - {(k0, result)} ∪ {(k0, v0)} ∧
             (result = null → ¬(∃v. (k0, v) ∈ old contents)) ∧
             (result ≠ null → (k0, result) ∈ old contents)" */
{ ... }
```

Figure 2-3: Method Contract for AssociationList.put()

instances of the same data structure. If these invariants were violated, then changes to one instance may inadvertently result in changes to another. By explicitly stating these constraints in the form of invariants, Jahob can ensure that the data structure operations preserve them. Without them, it may not be possible to verify that the data structure operates as specified, since the program code may implicitly depend on the validity of these constraints.

2.1.4 Method Contracts

Figure 2-3 presents the method contract for the put() method of the association list. In Jahob, a method contract consists of an ensures clause, and, optionally, a modifies clause, and/or a requires clause, depending on the method. The method contract for put() contains all three. The requires clause gives the method precondition. It states that the put() method must be invoked on a key-value pair that is non-null. The modifies clause gives the frame condition. It states that put() may modify the contents specification variable of the given association list (this..contents),[2] but does not modify any other public state. The ensures clause gives the method postcondition. It states that the effect of invoking put() is to add the given key-value pair to the contents of the association list. It additionally states that put() returns the previous binding of the given key, if any, and otherwise returns null.

2.1.5 Loop Invariants

In general, loop invariants are required to generate verification conditions for programs that contain loops. If the properties being verified are sufficiently simple, then it may be possible to automatically infer such invariants using program analyses or heuristics. But because Jahob is concerned with full functional correctness properties, which are, in general, undecidable, the loop invariants for our programs are typically provided by the developer in the form of annotations. In most cases, the system automatically identifies the variables whose values do not change across the body of the loop, so the annotated loop invariant need only capture the loop invariant properties for the variables whose values do change. Jahob also includes a shape

[2]Since put() is an instance method, Jahob automatically resolves the implicit reference to the receiver in the modifies clause.

```
private boolean _containsKey(Object k0)
/*: requires "theinvs"
    ensures "result = (∃v.((k0, v) ∈ contents)) ∧ theinvs" */
{
    Node current = first;
    while /*: inv "(∃v.(k0, v) ∈ contents) =
                   (∃v.(k0, v) ∈ current..con)" */
        (current != null) {

        if (current.key == k0)
            return true;

        current = current.next;
    }
    return false;
}
```

Figure 2-4: Loop Invariant in AssociationList._containsKey()

analysis engine Bohne [160] which can infer loop invariants, though we did not use Bohne for the verifications described in this thesis.

Figure 2-4 presents the _containsKey() method of the association list data structure. It is a private method that returns a boolean indicating whether a given key is mapped to a value in the association list. The loop invariant for _containsKey(), which is designated by the inv keyword, gives the following loop invariant property. It states that the key k0 is in the association list if and only if it is in the portion of the list reachable from the currently examined node (current).

2.2 Verification

Figure 2-5 describes Jahob's verification process. From the annotated program, Jahob produces verification conditions using a weakest liberal preconditions semantics. These verification conditions are higher-order logic formulas. If the formulas are proved correct, then the analyzed program is guaranteed to be correct with respect to the specification.

Jahob proves the correctness of these formulas using a diverse collection of internal and external automated reasoning systems—automated theorem provers, decision procedures, and program analyses. Jahob first translates each verification condition into an equivalent conjunct of smaller formulas. It then splits the conjunct into its component formulas, enabling the fine-grained application of different provers on a single verification condition. Jahob translates the resulting formulas into the logic subset appropriate for the given prover, using a sound formula approximation technique described in Chapter 3 (Section 3.3.3). The user selects the desired provers as a sequence on the command line. Jahob invokes the given sequence as a cascade on each formula. If the first prover fails to prove the given formula within a specified time-out, Jahob invokes the next prover in the sequence, and so on, until a prover is

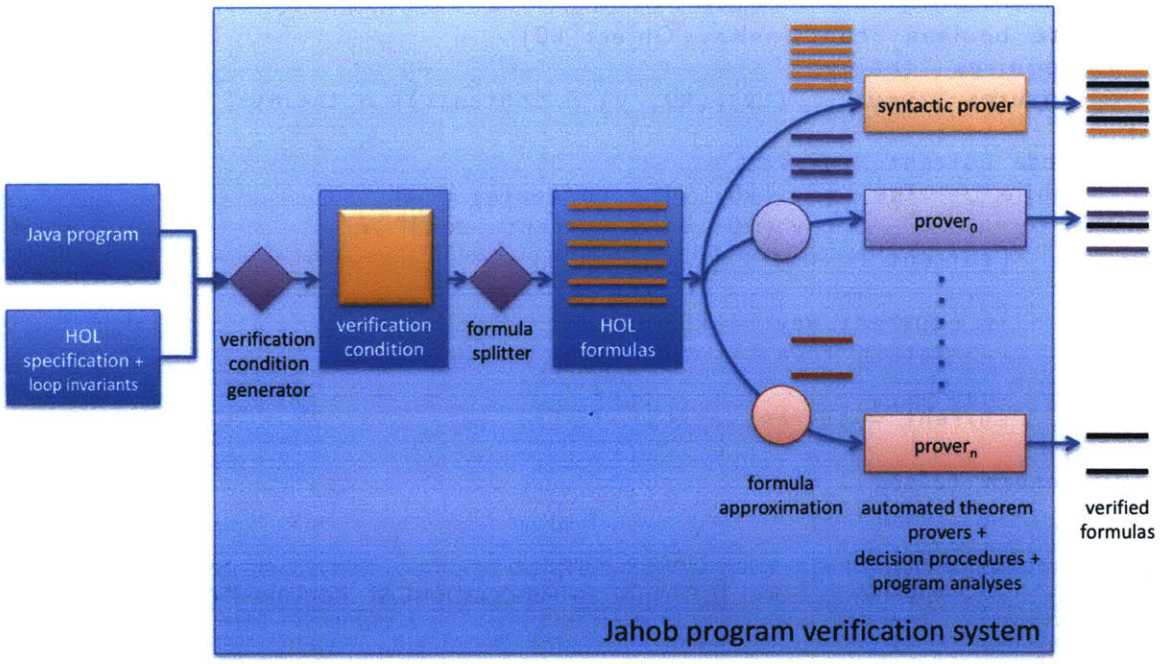


Figure 2-5: Jahob Verification Process

able to prove the given formula, or until the sequence of specified provers is exhausted. By default, Jahob processes the given formulas in sequence, but as each formula is completely independent, Jahob also includes an option for processing them in parallel. The verification succeeds if all the formulas are proved.

We verified the association list data structure using Jahob's internal syntactic prover, the SMT prover Z3 [116, 117], and the first-order prover SPASS [156]. Using these three provers, Jahob is able to completely verify the association list data structure with respect to its specification.

2.3 Integrated Proof Language

If the verification does not succeed, the developer can prove the failed formulas manually using interactive theorem provers. This is the standard approach used in most program verification systems. While Jahob includes interfaces to the proof assistants Isabelle [122] and Coq [24] for this purpose, this approach is challenging in practice. Interactive theorem provers require a great deal of expertise to use effectively. Because the formulas in question are mechanically generated, they tend to be large and heavily encoded, making them difficult to prove manually. Also, extracting these formulas from Jahob and proving them using an external proof assistant means that the developer cannot take advantage of the many diverse provers integrated into Jahob to assist with the proof.

Jahob's integrated proof language addresses these problems by allowing proof language commands to be embedded directly in the program. Developers can therefore prove failed formulas without leaving the familiar context of the program and program

```
public /*: claimedby Hashtable */ class Node {
    public Object key;
    public Object value;
    public Node next;
    /*: public ghost specvar con :: "(obj * obj) set" = "∅";
        ...
    */
}
public class Hashtable {
    private Node[] table = null;
    /*:
      public ghost specvar contents :: "(obj * obj) set" = "∅";
      public ghost specvar init :: "bool" = "False";

      static specvar h :: "(obj ⇒ int ⇒ int)";
      vardefs "h == (λo1. (λi1. ((abs (hashFunc o1)) mod i1)))";
      static specvar abs :: "(int ⇒ int)"
      vardefs "abs == (λi1. (if (i1 < 0) then (−i1) else i1))";

      invariant ContentsDefInv: "init →
      contents = {(k,v). (k,v) ∈ table.[(h k (table..length))]..con}";
      invariant Coherence: "init → (∀i k v. 0 ≤ i ∧ i < table..length →
      (k,v) ∈ table.[i]..con → h k (table..length) = i)";
      ...
    */
    ...
}
```

Figure 2-6: Hashtable Example

verification system. Like Jahob specification constructs, the proof commands are annotations within the program that use Jahob's formula language. Proof commands can refer to the current program state using the same concrete and specification variables used by the Java program and specification constructs, avoiding the problem of manipulating heavily encoded formulas in external proof assistants. The commands direct Jahob's combined automated reasoning system. This allows developers to provide only the minimum amount of guidance that the provers need to successfully complete the proof, avoiding the many manual proof steps that may be necessary when proving the same formula using a proof assistant.

2.3.1 Hash Table Example

Figure 2-6 presents an excerpt from a hash table data structure that we will use to illustrate some of our proof language commands. Chapter 7 presents the hash table in more detail. The hash table data structure is implemented as an array of buckets, where each bucket consists of a singly-linked list of Node objects. The abstract state of the hash table is very similar to that of the association list. The contents specification variable in the Hashtable class characterizes the mappings in the hash table as a set of key-value pairs. The con specification variable of the Node class characterizes the

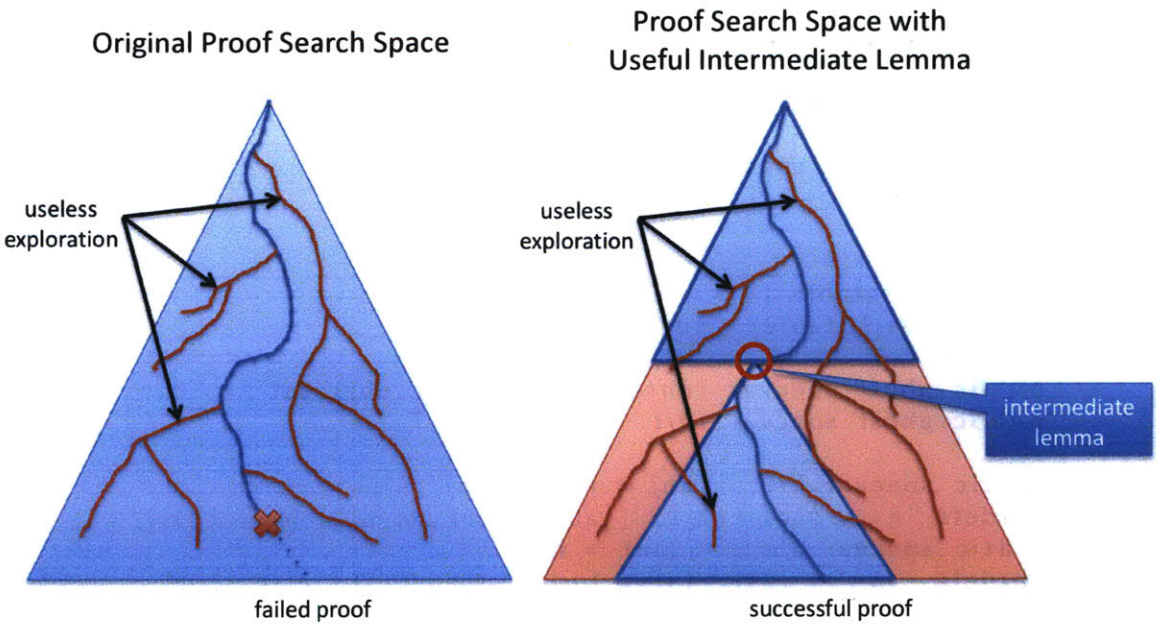


Figure 2-7: Effect of Useful Intermediate Lemma on Proof Search Space

mappings reachable from a given node as a set of key-value pairs. The abstract state of the hash table also includes a specification variable init, which is a boolean variable that is true if the hash table is initialized, and false otherwise.

The ContentsDefInv invariant gives the definition for contents. It states that, for initialized hash tables, the contents of the hash table consist of the key-value pairs stored in the buckets, considering only those key-value pairs that are mapped to the correct buckets. The Coherence invariant additionally requires that all key-value pairs in the hash table be mapped to the correct buckets. The static specification variable h is a shorthand for the hash function, which is given by the corresponding vardefs definition. The static specification variable abs is a shorthand for the absolute value function. The hash table data structure also contains additional invariants, which we omit here for clarity.

2.3.2 Identifying Intermediate Lemmas

The most common use of Jahob's proof language is for directing the combined automated reasoning system in proving an intermediate lemma. In general, this would be an intermediate lemma that is necessary to the successful proof of a desired formula. The proof command that makes this possible is the note command.

There are two scenarios in which it is helpful to identify a useful intermediate lemma for the system. First, if the proof of a formula is sufficiently complex, the provers may not be able to find a proof in a reasonable amount of time. Figure 2-7 illustrates this scenario. Without a useful intermediate lemma, the prover needs to traverse a large proof search space to find the proof. The size of the search space may cause the prover to spend so much time exploring unproductive paths that the proof fails. Identifying a useful intermediate lemma divides the original complex proof

```
1  public Object get(Object k0)
2  /*: requires "init ∧ k0 ≠ null"
3      ensures "(result ≠ null → (k0, result) ∈ contents) ∧
4               (result = null → ¬(∃ v. (k0, v) ∈ contents))" */
5  {
6      //: instantiate "theinv ContentsDefInv" with "this";
7      /*: mp ThisContentsDef:
8          "this ∈ alloc ∧ this ∈ Hashtable ∧ init → contents =
9           {(k, v). (k, v) ∈ table.[(h k (table..length))]..con}"; */
10
11     int hc = compute_hash(k0);
12     Node curr = table[hc];
13
14     //: note HCDef: "hc = h k0 (table..length)";
15     /*: note InCurr: "∀v.(((k0, v) ∈ contents) = ((k0, v) ∈ curr..con))"
16         from ThisContentsDef, HCDef; */
17
18     while /*: inv "∀v.((k0, v) ∈ contents) = ((k0, v) ∈ curr..con)" */
19         (curr != null) {
20
21         if (curr.key == k0)
22             return curr.value;
23
24         curr = curr.next;
25     }
26     return null;
27 }
```

Figure 2-8: Proof Commands in Hashtable.get()

task into two simpler tasks. The first is that of finding a proof for the intermediate lemma. The second is that of finding a proof for the original goal from the useful intermediate lemma and other available facts. This division effectively reduces the proof search space. As Figure 2-7 illustrates, this reduction can often enable a prover to successfully prove a formula that it is unable to prove without the relevant lemma.

The second situation in which it is helpful to identify an intermediate lemma is when the proof of a formula requires the expertise of more than one specialized prover. In this case, a note command can identify a useful intermediate lemma that can be proved by a single prover. Once that lemma is proved, it can then be used by a different prover to prove the original formula. Where Jahob's splitting process is able to divide large formulas into smaller ones syntactically for the fine-grained application of different specialized provers, the note command is able to divide complex formulas semantically.

Figure 2-8 presents the get() method of the hash table data structure. The get() method takes a key k0, and returns the value corresponding to k0 in the hash table. If the hash table does not contain a mapping for k0, get() returns null. The get() method does this by first computing the index for the bucket to which k0 is hashed. It then searches through the linked list at that bucket until it either finds k0, or reaches the

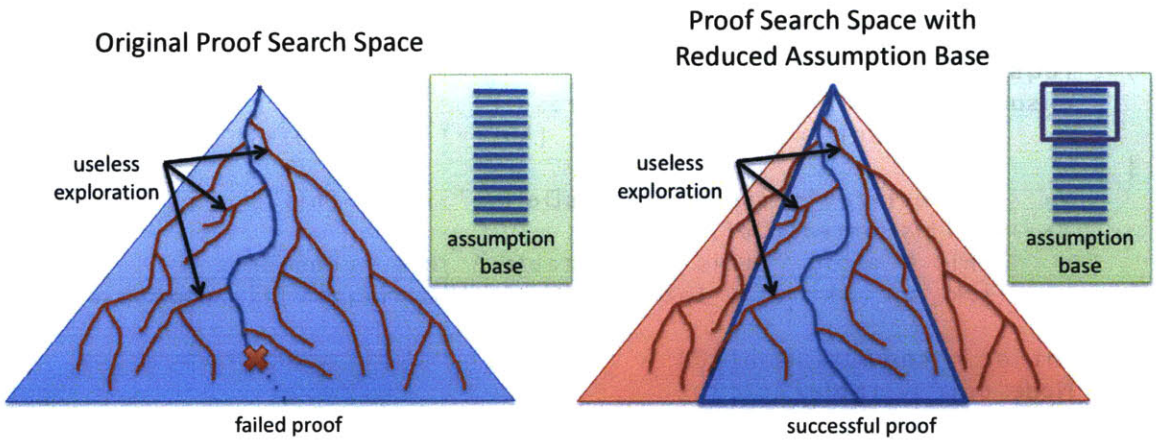


Figure 2-9: Effect of Controlling Assumption Base on Proof Search Space

end of the list. The while loop that searches through the bucket is annotated with a loop invariant. It states that the key k0 has a mapping in the hash table if and only if it has a mapping in the list reachable from the currently searched node curr.

Jahob verifies loops by checking that the loop invariant holds for both the base and inductive cases. In the get() method, the base case—proving that the loop invariant holds on the initial entry into the loop—does not verify automatically. The proof commands in the method therefore guide the combined reasoning system to prove this case. We use several proof commands to do this, including two note commands.

The note command in line 14 identifies an intermediate lemma that is useful for the proof of the base case for the loop invariant. This lemma states that the local variable hc corresponds to correct index for the bucket to which k0 is hashed. The effect of the note command is to direct Jahob to prove the intermediate lemma, and to name it HCDef. Once proved, the lemma is available for use in proofs of subsequent formulas. In this case, it is used in the following note command to prove the desired loop invariant.

2.3.3 Controlling the Assumption Base

When Jahob presents a formula to an automated reasoning system to prove, it does so in the form of a sequent. Given a sequent $F_1, \ldots, F_n \vdash G$, the formulas $F_1, \ldots, F_n$ are the facts that are available to the prover when proving G. We refer to this set of facts as the *assumption base*. The default assumption base for a formula contains all the known facts about the program at a given program point. This is because the system has no way of knowing which facts are relevant to the proof of that particular formula. But when the assumption base is large, the prover has to deal with a correspondingly large proof search space, which can make it difficult for the prover to find a proof. In some cases, the presence of unnecessary facts in the assumption base can cause a prover to fail to prove a formula that it is otherwise able to prove, when presented with only the relevant facts. Figure 2-9 illustrates this scenario.

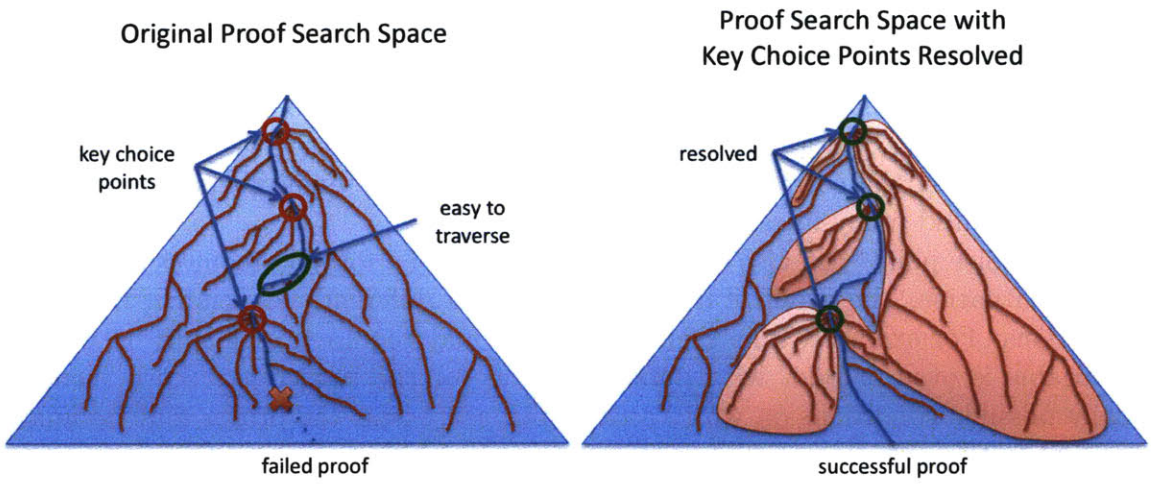


Figure 2-10: Effect of Resolving Key Choice Points on Proof Search Space

Jahob's proof language addresses this problem by giving developers control over the assumption base using a *from clause*. The from clause is an optional component of a proof command that allows the developer to specify, by name, the set of facts to use in proving a formula. These facts can be named lemmas that previous proof commands have proved, or known facts about the current program state—such as class invariants, or method preconditions—to which Jahob assigns default names. By reducing the size of the assumption base that Jahob passes to the provers, the from clause effectively reduces the size of the proof search space. As Figure 2-9 illustrates, this reduction is often sufficient to enable a prover to prove a formula that it is unable to prove using the original, larger assumption base.

In Figure 2-8, the note command in line 15 contains a from clause that directs the system to prove the loop invariant using the named facts ThisContentsDef and HCDef. ThisContentsDef is a named lemma that the proof command at line 7 proves, while HCDef is the named lemma that the previously discussed note command proves.

2.3.4 Resolving Key Choice Points

In some cases, the proof search space for a given formula can be large even when the assumption base is not. This occurs when there are *key choice points* in the proof search space. Key choice points occur when the prover has to make a decision between a potentially unbounded number of alternatives. Figure 2-10, which illustrates this scenario, shows how some parts of the proof search space have few branching paths, and are therefore easy for a prover to traverse, while others contain many branching paths due to key choice points. The inability of a prover to make a correct decision at a key choice point and the time needed to explore the many incorrect alternatives can cause the prover to fail to find a proof.

Key choice points occur for several reasons. First, they can occur when there are universally quantified formulas in the assumption base (i.e. formulas of the form $\forall \vec{x}.F$). In this case, the prover has an unbounded number of terms with which it can

instantiate the universally quantified assumption. This can cause a blow up in the proof search space, which can then result in a failed proof. Another situation in which key choice points can occur is when the prover is attempting to prove an existentially quantified goal (i.e. a formula of the form $\exists \vec{x}.G$). In this case, the prover may have difficulty finding the correct witness for the existentially quantified formula among the (potentially) many incorrect ones.

Key choice points can also occur when the proof of a formula requires a case split. In this situation, the proof of a formula proceeds differently depending on certain conditions. For example, a mathematical proof may require a case split on whether a variable is even or odd, with different proofs for the two cases. Since there are many possible conditions on which a case split can occur, the prover may be unable to identify the correct case split, or to even recognize that a case split is necessary.

Jahob's proof language addresses these problems by supporting proof commands that allow developers to resolve key choice points for the combined reasoning system. The instantiate command allows the developer to identify the correct term with which to instantiate a universally quantified assumption. The witness command allows the developer to identify the correct witness for an existentially quantified goal. The cases command allows the developer to specify the correct case split for a proof. By resolving key choice points, these commands effectively reduce the size of the proof search space. As Figure 2-10 illustrates, this reduction is often sufficient to enable a prover to find a proof for a formula that it was previously unable to prove.

In Figure 2-8, the instantiate command in line 6 directs Jahob to instantiate the ContentsDefInv invariant with the receiver object. The effect of the command is to direct Jahob to first ensure that ContentsDefInv holds, then add the instantiated formula to the assumption base, making it available in the verification of subsequent formulas. By doing so, it provides the provers with a fact needed for subsequent proofs, while avoiding the need for the provers to search through an unbounded number of potential terms.

2.3.5 Decomposing Complex Goals

To appropriately identify useful intermediate lemmas and resolve key choice points, it is sometimes necessary to decompose complex proof goals. For example, if the goal is a universally quantified formula $\forall \vec{x}.G$, it can be difficult to state the relevant intermediate lemmas or resolve key choice points without referring to the universally quantified variables $\vec{x}$. If the goal is an implication $F \rightarrow G$, it may be desirable to temporarily assume that F holds, prove intermediate lemmas that are consequences of F, then prove G from those lemmas. In both cases, the provers need access to components of the goal, making it desirable to have proof language commands that allow the proof goal to be decomposed appropriately.

The pickAny and assuming commands are the proof language commands that enable this decomposition. The pickAny command creates a hypothetical block in which the universally quantified variables are free, making it possible to write proof commands within the block that refer to the universally quantified variables. The assuming command creates a hypothetical block in which the antecedent of the implication

```
{
    /*: pickAny ht::obj suchThat
        ContentsDefHyp: "ht ∈ alloc ∧ ht ∈ Hashtable ∧ ht..init"; */
    /*: note ContentsThis: "ht = this → ht..contents =
        {(k,v). (k,v) ∈ ht..table.[(h k (ht..table..length))]..con}"
        from OldContents, ElementInjInv, Acyclic, ThisProps, KFound,
        VFound, ConDef, FProps, FNonNull, HashInv, HCProps; */
    ...
    /*: note ContentsDefPostCond: "ht..contents = {(k, v).
        (k, v) ∈ ht..table.[(h k (ht..table..length))]..con}"
        from ContentCases forSuch ht; */
}
```

Figure 2-11: Proof Commands from Hashtable.removeFirst()

holds. This makes it possible to derive intermediate lemmas from the antecedent, then use those lemmas to prove the consequent of a goal that is an implication. The pickAny command also supports a shorthand for decomposing a universally quantified implication, for use in place of a pickAny block composed with an assuming block. In general, commands that decompose complex proof goals are some of the most straightforward commands to use, since they follow directly from the structure of the failed formula.

Figure 2-11 presents an excerpt from a pickAny block in the Hashtable.removeFirst() method, which is described in more detail in Chapter 7 (see Section 7.6.7). The proof commands in this block direct Jahob in proving the ContentsDefInv for the postcondition of the removeFirst() method, a private method in the hash table data structure used to remove the first node in a bucket. The pickAny command creates a hypothetical block in which the universally quantified variable in the ContentsDefInv invariant corresponds to the free variable ht. Within this block, the note commands can therefore refer to ht and state intermediate lemmas in which ht is free. The net effect of the commands in Figure 2-11 is to prove that the ContentsDefInv holds, in the form of the following formula:

$$\forall \mathsf{ht}.\mathsf{ht} \in \mathsf{alloc} \wedge \mathsf{ht} \in \mathsf{Hashtable} \wedge \mathsf{ht..init} \rightarrow$$
$$\mathsf{ht..contents} = \{(\mathsf{k},\mathsf{v}).(\mathsf{k},\mathsf{v}) \in \mathsf{ht..table.[(h\ k\ (ht..table..length))]..con}\}$$

The pickAny command additionally adds the proved invariant into the assumption base, and gives it the name ContentsDefPostCond, making it available for the verification of subsequent formulas.

2.3.6 Other Proof Commands

Jahob also supports other proof commands that are described in more detail in Chapter 4. These include the proof commands we've described here, as well as commands that encode first-order logic rules of deduction, induction, and commands for controlling the assumption base. Together, these proof commands make it possible for us to

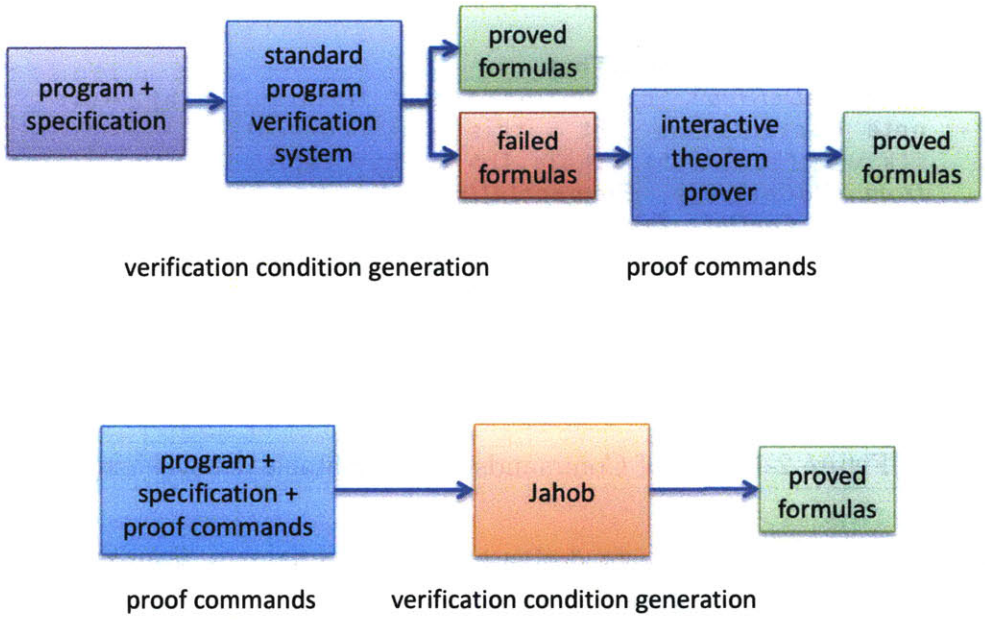


Figure 2-12: Jahob Proof Commands versus Standard Program Verification Approach

verify the full functional correctness of our data structures without the use of external proof assistants, even for complex data structures like hash tables, priority queues, and binary search trees. Using Jahob's integrated proof language, the internal syntactic prover, Z3, SPASS, and Jahob's automated tactic interface to Isabelle [122], we are able to prove the correctness of the above hash table data structure with respect to its specification.

2.3.7 Why the Proof Language Works

The key insight that makes Jahob's integrated proof language possible is the observation that proof commands can, in effect, be pushed backwards through the weakest liberal precondition mechanism that Jahob uses to produce verification conditions. Standard program verification systems generate formulas that only tell the provers what must be proved to verify the correctness of the program. If a formula fails to prove, the developer must prove it manually at the level of the failed verification condition formula, as illustrated in Figure 2-12. Through the proof language, Jahob not only tells the provers what to prove, but also how to prove it. In this way, developers are able to control the proof of complex formulas at the program level, instead of manually proving failed formulas post verification condition generation. Figure 2-12 illustrates this difference.

2.3.8 Practical Advantages of the Proof Language

When proving failed formulas post verification condition generation, the formulas involved are the result of transforming the program and specifications through the internal processes of the program verification system. These formulas tend to be large, heavily encoded, and hence, difficult to understand and prove. In Jahob, the proof commands are embedded directly in the program. The formulas involved are at the program level, and have not been subject to the internal processes of the program verification system. As a result, they are much easier to understand and verify. Not only that, but the effect of the proof commands is to generate verification condition formulas that can be automatically proved using the automated provers integrated into the system. This reduces the need for specifying detailed proof steps, since the provers automate a large part of the proofs.

2.4 Summary

This chapter uses excerpts from verified data structures to illustrate the Jahob specification constructs and integrated proof language. The excerpts are taken from the verified association list and hash table data structures, which are described in more detail in Chapters 6 and 7, respectively.

We use the association list data structure to illustrate the use of Jahob specification constructs. Jahob specifications, which take the form of annotations within the code, describe the desired behavior of the program using specification variables, class invariants, and method contracts. Specification variables characterize the data structure's abstract state, providing a mechanism for describing the behavior of the data structure without revealing the underlying representation. Class invariants and vardefs declarations give the abstraction function relating the abstract and concrete state. Class invariants may also state additional constraints on valid program states that ensure the correct operation of the data structure. Method contracts describe the desired behavior of methods. From the annotated program, Jahob generates verification condition formulas, and proves them using a diverse collection of internal and external automated reasoning systems. If all the formulas prove, then the program is guaranteed to be correct with respect to its specification. Jahob is able to verify the correctness of the association list data structure using Jahob's internal syntactic prover, the SMT prover Z3, and the first-order prover SPASS.

If the formulas fail to prove, Jahob's integrated proof language allows the developer to guide the provers to successfully verify the correctness of the program. We illustrate the use of the integrated proof language using the hash table data structure. The proof language includes commands such as note for identifying useful intermediate goals to the combined reasoning system, optional constructs such as the from clause for controlling the assumption base used to prove a formula, commands such as instantiate for resolving key choice points in the prover's proof search space, and commands such as pickAny for decomposing complex goals. The proof commands direct the efforts of the automated reasoning systems in verifying properties that would

otherwise be beyond their capabilities to prove. The key insight that makes this possible is the observation that proof commands can, in effect, be pushed backwards through the weakest liberal precondition mechanism that Jahob uses to produce verification conditions. In standard program verification practice, developers must proved failed verification condition formulas manually using external proof assistants. But because these formulas are the result of applying many transformations to the original annotated program, they are often large, heavily encoded, and, hence, difficult to understand and prove. By allowing developers to embed proof commands directly in the program, the integrated proof language makes it possible to work with formulas at the level of the program, which are much easier to understand and prove. The net effect of the proof commands is to produce verification condition formulas that can be proved automatically using the integrated provers, further simplifying the proof task. Using Jahob's integrated proof language, the internal syntactic prover, Z3, SPASS, and Jahob's automated tactic interface to Isabelle, we are able to prove the correctness of the hash table data structure with respect to its specification.

Chapter 3

The Jahob System

Jahob is based on the standard specification and verification approach [58], in which verification condition formulas are generated from programs annotated with specifications. The verification conditions are then proved, either manually or using automated techniques, to ensure the correctness of the program. This approach is also the basis of many other program verification systems including ESC/Modula-3 [51], ESC/-Java [56], ESC/Java2 [39, 77], Spec# [11, 12], KeY [16], and Krakatoa/Why [55, 106].

Jahob differs from these systems in its use of integrated reasoning and an integrated proof language. Integrated reasoning enables the application of a diverse collection of automated reasoning systems on the generated verification conditions, while the integrated proof language extends the standard specification and verification approach to enable developers to specify proof strategies to the program verification system. In this chapter, we describe the Jahob system, including the subset of Java that Jahob supports, our specification language, verification condition generation, the techniques we use to implement integrated reasoning, as well as the automated provers, analyses, and decision procedures to which Jahob interfaces. We discuss Jahob's integrated proof language in Chapter 4.

3.1 Jahob Programs

Figures 3-1, 3-2, and 3-3 describe the format for Jahob programs. Jahob verifies Java programs annotated with specifications and proof commands written as special Java comments of the form /*: ... */ and //: ..., enabling the use of standard Java compilers and virtual machines on annotated programs. The subset of Java supported includes classes, methods, objects, fields, arrays, loops, conditional statements, and integer and boolean operations. Jahob interprets these constructs according to standard Java semantics [61]. Our implementation also supports common Java features such as the implicit use of the receiver parameter this, automatic disambiguation of field references and operator precedence according to standard scoping rules, multiple levels of array and field dereferences, and program comments, as well as a number of useful shorthands specific to Jahob specifications. For clarity, we have omitted these features from Figures 3-1, 3-2, and 3-3, and focused on a core set of valid Jahob pro-

grams. The current implementation of Jahob does not support dynamic class loading, exceptions,[1] or inheritance, though techniques exist that should make it possible to extend Jahob with support for such constructs [68, 11, 77].

Specifications for Jahob programs are written in terms of specification variable declarations, method contracts, class invariants, loop invariants, and annotations within method bodies. Many of these specification constructs contain higher-order logic formulas that capture properties of the program state. In this section, we describe Jahob formulas as well as the details of each of the supported specification constructs.

int	:	*Integer*
bool	:	*Boolean*
class_decl	::=	*access claim* class *class_name* {*decls*}
access	::=	public | private | ϵ
claim	::=	/*: claimedby *class_name* */ | ϵ
decls	::=	*decl decls* | ϵ
decl	::=	*field_decl* | *class_spec* | *method_decl*
field_decl	::=	*access static java_type field_name*;
	|	*access static java_type field_name* = *const*;
static	::=	static | ϵ
java_type	::=	int | boolean | *class_name* | *class_name*[]
const	::=	*int* | *bool* | null
class_spec	::=	/*: *_class_spec_* */
*_class_spec_*	::=	*access static* invariant *inv_name* : "*form*"
	|	*access static ghost* specvar *specvar_name* :: "*type*"
	|	*access static* ghost specvar *specvar_name* :: "*type*" = "*form*"
	|	*access* vardefs "*specvar_name* == *form*"
ghost	::=	ghost | ϵ
type	::=	obj | bool | int | (*type* set) | (*type* $*$ *type*) | (*type* $\Rightarrow$ *type*)

Figure 3-1: Jahob Programs

3.1.1 Jahob Formulas

Jahob formulas follow the grammar given in Figure 3-3. The syntax and semantics of these formulas are the same as that of formulas in Isabelle/HOL [122]. Formulas are simply typed with ground types bool for boolean values, int for integers, and obj for objects, as well as type constructors $\Rightarrow$ for total functions, $*$ for tuples, and set for sets. The logic contains polymorphic equality, the standard logical connectives $\wedge$, $\vee$,

[1] In verified data structures adapted from java.util, where the original java.util implementation throws an exception, the Jahob implementation generally uses preconditions to guard against the exceptional behavior.

$$
\begin{array}{rcl}
\mathit{method\_decl} & ::= & \mathit{access}\ \mathit{ret\_type}\ \mathit{method\_name}\ (\mathit{params})\ \mathit{method\_spec} \\
 & & \{\mathit{stmts}\} \\
\mathit{ret\_type} & ::= & \mathsf{void} \mid \mathit{java\_type} \\
\mathit{params} & ::= & \mathit{java\_type}\ \mathit{param\_name},\ \mathit{params} \mid \epsilon \\
\mathit{method\_spec} & ::= & /\!*\text{: } \mathit{requires}\ \mathit{modifies}\ \mathit{ensures}\ *\!/ \\
\mathit{requires} & ::= & \mathsf{requires}\ \text{“}\mathit{form}\text{”} \mid \epsilon \\
\mathit{modifies} & ::= & \mathsf{modifies}\ \mathit{mods} \mid \epsilon \\
\mathit{ensures} & ::= & \mathsf{ensures}\ \text{“}\mathit{form}\text{”} \\
\mathit{mods} & ::= & \mathit{mod}\ \mathit{mods} \mid \mathit{mod} \\
\mathit{mod} & ::= & \text{“}\mathit{\_mod\_}\text{”} \\
\mathit{\_mod\_} & ::= & \mathit{field} \mid \mathit{var\_name}..\mathit{field} \\
 & \mid & \mathsf{new}..\mathit{field} \mid \mathsf{new}..\mathsf{arrayState} \mid \mathsf{arrayState} \\
\mathit{field} & ::= & \mathit{field\_name} \mid \mathit{class\_name}.\mathit{field\_name} \\
\mathit{stmts} & ::= & \mathit{stmt}\ \mathit{stmts} \mid \epsilon \\
\mathit{stmt} & ::= & \mathit{java\_type}\ \mathit{var\_name};\ \mid \mathit{java\_type}\ \mathit{var\_name} = \mathit{expr}; \\
 & \mid & \mathit{var\_name} = \mathit{expr}; \\
 & \mid & \mathit{field\_expr} = \mathit{expr};\ \mid \mathit{array\_expr} = \mathit{expr}; \\
 & \mid & \mathsf{while}\ /\!*\text{: } \mathsf{inv}\ \text{“}\mathit{form}\text{”}\ *\!/\ (\mathit{expr})\ \{\mathit{stmts}\} \\
 & \mid & \mathsf{if}\ (\mathit{expr})\ \{\mathit{stmts}\}\ \mathsf{else}\ \{\mathit{stmts}\} \\
 & \mid & \mathit{proc\_call};\ \mid \mathsf{return}\ \mathit{expr}; \\
 & \mid & /\!*\text{: } \mathit{proof\_cmd}\ *\!/\ \mid /\!*\text{: } \mathit{spec\_stmt}\ *\!/ \\
\mathit{field\_expr} & ::= & \mathit{field} \mid \mathit{var\_name}.\mathit{field} \\
\mathit{array\_expr} & ::= & \mathit{var\_name}[\mathit{expr}] \\
\mathit{proc\_call} & ::= & \mathit{method\_name}\ (\mathit{exprs}) \\
\mathit{exprs} & ::= & \mathit{\_exprs\_} \mid \epsilon \\
\mathit{\_exprs\_} & ::= & \mathit{expr} \mid \mathit{expr}, \mathit{\_exprs\_} \\
\mathit{expr} & ::= & \mathit{var\_name} \mid \mathit{field\_expr} \mid \mathit{array\_expr} \\
 & \mid & \mathsf{new}\ \mathit{class\_name}(\mathit{exprs}) \mid \mathsf{new}\ \mathit{class\_name}[\mathit{expr}] \mid \mathit{const} \\
 & \mid & \mathit{proc\_call} \\
 & \mid & !\mathit{expr} \mid -\mathit{expr} \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \mid (\mathit{expr}) \\
\mathit{op} & ::= & + \mid - \mid * \mid / \mid \% \mid == \mid != \mid \&\& \mid || \\
\mathit{spec\_stmt} & ::= & \mathsf{ghost\ specvar}\ \mathit{specvar\_name} :: \text{“}\mathit{type}\text{”} \\
 & \mid & \mathsf{ghost\ specvar}\ \mathit{specvar\_name} :: \text{“}\mathit{type}\text{”} = \text{“}\mathit{form}\text{”} \\
 & \mid & \text{“}\mathit{specvar}\text{”} := \text{“}\mathit{form}\text{”} \\
 & \mid & \text{“}\mathit{var\_name}..\mathit{specvar}\text{”} := \text{“}\mathit{form}\text{”} \\
 & \mid & \mathsf{havoc}\ \mathit{var\_name}\ \mathsf{suchThat}\ l\text{: “}\mathit{form}\text{”} \\
\mathit{specvar} & ::= & \mathit{specvar\_name} \mid \mathit{class\_name}.\mathit{specvar\_name}
\end{array}
$$

Figure 3-2: Jahob Methods

$$
\begin{array}{rcl}
annot & : & String \\
F, G & : & form \\
f & : & field \\
form & ::= & unop\ G \mid F_1\ binop\ F_2 \mid [\![F_1; \ldots; F_n]\!] \\
& \mid & \forall x.G \mid \exists x.G \mid \lambda x.G \mid \{x.G\} \mid \{F_1, \ldots, F_n\} \mid (F_1, \ldots, F_n) \\
& \mid & F_{obj}..F_{fld} \mid F_{fld}\ F_{obj} \mid F_{fld}(F_{obj} := F_{src}) \\
& \mid & F_{arr}.[F_{ind}] \mid F_{arrSt}\ F_{arr}\ F_{ind} \mid (F_{arrSt}\ F_{arr})(F_{ind} := F_{src}) \\
& \mid & \mathsf{cardinality}(form) \mid \mathsf{old}\ form \mid (u, v) \in \{(x, y).G\}^* \mid \mathsf{tree}[f_1, \ldots, f_n] \\
& \mid & \emptyset \mid \mathsf{arrayState} \mid \mathsf{alloc} \mid \mathsf{hidden} \mid var\_name \mid field \mid const \\
& \mid & \mathsf{comment}\ \text{"}annot\text{"}\ G \mid G :: type \mid (G) \\
unop & ::= & - \mid \neg \\
binop & ::= & = \mid \neq \mid == \mid \vee \mid \wedge \mid \rightarrow \mid \Longrightarrow \\
& \mid & \in \mid \cap \mid \cup \mid - \mid + \mid * \mid / \mid \% \mid < \mid > \mid \leq \mid \geq
\end{array}
$$

Figure 3-3: Jahob Formulas

$\neg$, $\rightarrow$, $\forall$, $\exists$, as well as the λ binder, set comprehension $\{e.F\}$, and standard operations on sets and integers. It supports selected defined operations, most notably $(u, v) \in \{(x, y).G\}^*$ for transitive closure, $\mathsf{tree}[f_1, \ldots, f_n]$ denoting that a data structure is a tree, and cardinality (card for short) for the cardinality of finite sets. Formulas can be annotated with comments using the keyword comment, as in comment "c" F, which annotates the formula F with comment c. The shorthands $F_{obj}..F_{fld}$ and $F_{arr}.[F_{ind}]$ stand for the field and array dereference notations $F_{fld}\ F_{obj}$ and $F_{arrSt}\ F_{arr}\ F_{ind}$, respectively.

For annotations occurring within the program code, if an annotation at program point p contains a formula F, then an occurrence of a program variable v in F denotes the value of v at p. The old operator changes this interpretation: old v denotes the value of the variable v at the entry of the currently analyzed method. To denote values of variables at other program points, developers can use specification variables to save these values.

Jahob formulas may also refer to special Jahob keywords such as arrayState, which refers to the array component of the global program state, alloc, the set of all allocated objects, and hidden, the set of objects that are private to a given class. Sections 3.2.1 and 3.2.2 discuss the use of these keywords and the meaning of the corresponding formula expressions in terms of Java and specification constructs.

3.1.2 Specification Variables

Specification variables are abstract variables that do not exist during program execution, but are used to represent the abstract state of the program, so that we can describe the behavior of the program without revealing the underlying data representation [56, Section 4]. In Jahob, developers use the specvar keyword to declare a specification variable, indicate its type, an optional initial value, and whether the

variable is public or private, static or instance, or a *ghost variable*. For instance specification variables, Jahob lifts the variable's type from the specified type t to obj $\Rightarrow t$, converting it into a variable of function type. A ghost variable is a type of specification variable that must be updated explicitly by the developer (using a specification assignment statement) for its value to change, with Jahob ensuring the soundness of such updates. A dependent variable (also known as a *defined variable* [163]) is designated by the absence of the ghost keyword in its declaration, and is simply a way to name the value of an expression. Developers use the vardefs keyword followed by a definition of the form "$v == e$" to define the value of a specification variable v, where e is a formula that may contain occurrences of other variables. At any program point, the value of v is equal to the value of e at that program point. To ensure that dependent variables are well-defined, Jahob requires that their definitions be acyclic. For recursive definitions, the developer can use either transitive closure or a ghost variable with a class invariant that encodes the desired recursive relationship.

3.1.3 Method Contracts

Method contracts in Jahob consist of three parts: 1) a precondition, or requires clause, stating the properties of the program state and parameter values that must hold before the method is invoked; 2) a frame condition, written as a modifies clause, listing the components of the state that the method may modify, while the remaining components remain unchanged; and 3) a postcondition, or ensures clause, describing the state at the end of the method (possibly defined relative to the parameters and state at the entry of the method). Jahob uses method contracts for assume-guarantee reasoning in the standard way. When analyzing a method m, Jahob assumes m's precondition and checks that m satisfies its postcondition and the frame condition. Dually, when analyzing a call to m, Jahob checks that the precondition of m holds and assumes that the values of state components from the frame condition of m change subject only to the postcondition of m, and that the state components not in the frame condition of m remain unchanged. Method contracts of public methods omit changes to the private state of their enclosing class and instead use public specification variables to describe how they change the state. In most cases, Jahob does not require method contracts to specify changes to newly allocated objects. The exception is if a field f is changed for allocated objects and is otherwise not mentioned in the modifies clause, then the developer needs to add the special item new..f to the modifies clause.

3.1.4 Class Invariants

In Jahob, a class invariant can be thought of as a boolean-valued specification variable that Jahob implicitly conjoins with the preconditions and postconditions of every public method of the class. Class invariants are declared using the invariant keyword, an optional label, and a Jahob formula that captures the desired invariant. Developers can declare an invariant as private or public (the default annotation is private), but typically, a class invariant is private and is visible only inside the implementation of the class. Jahob conjoins the class invariants of a class C to the preconditions

and postconditions of public methods declared in C when verifying that methods in C respect their contracts. At method invocations, Jahob conjoins the public class invariants of the callee to the method preconditions and postconditions. To ensure soundness in the presence of callbacks, Jahob also conjoins private class invariants of C to each reentrant call to a method m declared in a different class C_1. This policy ensures that the invariant C will hold if $C_1.m$ (either directly or indirectly) invokes a method in C. To make an invariant hold less often than given by this policy, the developer can specify the invariant as $b \rightarrow F$ for some specification variable b and the desired property F, using b as a flag to control when F should hold. To make an invariant I with label l hold more often, the developer can use assertions with the shorthand (theinv l) that expand into I. This shorthand can be used in any Jahob formula, including those in method contracts and loop invariants, to refer to invariants in scope.

3.1.5 Annotations Within Method Bodies

Jahob supports several different types of annotations within method bodies to refine expectations about the behavior of the code and to debug the verification process. Specifically, developers can use these annotations to designate loop invariants, declare and update specification variables, and add to or release the system from the default proof obligations. Here we describe the supported annotations.

Loop Invariants

Jahob's verification condition generation algorithm requires loop invariants, which are typically provided by the developer. The loop invariant for a loop should appear immediately following the while keyword, and is designated by the keyword invariant (inv for short). A loop invariant must hold on entry to the loop (i.e. before the loop condition) and must be preserved by each iteration of the loop. The developer can omit conditions that depend only on variables not modified in the loop, as Jahob uses a simple syntactic analysis to conclude that the loop preserves such conditions.

Jahob also supports loop invariant inference using Bohne [160] (though we did not make use of this support for the data structures verified in this thesis). Bohne uses a technique based on symbolic shape analysis, and is able to infer complex loop invariants involving reachability and universal quantifiers.

Local Specification Variables

Local specification variables are similar to specification variables at the class level, but are local to a particular method. Developers can introduce local ghost and dependent specification variables by declaring them within the method body. Such variables can be helpful for simplifying proof obligations and for stating relationships between the values of variables at different program points.

Specification Assignment Statements

Specification assignment statements enable developers to change the value of a ghost variable. These assignment statements have the form $v := e$, where v is the name of the ghost variable being assigned, and e a formula that may contain occurrences of variables including v (the value of v in e in that case is its value before the assignment, as in assignment statements for concrete variables). Jahob also supports specification assignments of the form $x..f := e$, which is a shorthand for $f := f(x := e)$. Here $f(x := e)$ is the standard function update expression returning a function identical to f except at x where it has value e. To ensure that the verified program is consistent with its unannotated counterpart, specification assignment statements may not modify the concrete state.

Non-Deterministic Change

Jahob also supports a mechanism for assigning a value to a specification variable non-deterministically. An annotation of the form havoc x suchThat G, where x is a specification variable and G is a formula, changes the value of x subject only to the constraint G (for example, havoc x suchThat $0 \leq x$ sets x to an arbitrary non-negative value). To ensure soundness, Jahob emits an assertion that verifies that at least one such value of x exists. Consequently, havoc can also be used to "pick a witness" for an existentially quantified property $\exists x.G$ and to make this witness available for subsequent specification. A specification assignment of the form $x := e$ (for x not occurring in e) is a special case of a havoc statement whose condition is $x = e$ (its feasibility condition is trivial).

Assert

Jahob's assertion mechanism allows the developer to add static checks to the body of a method. An assert G annotation at program point p requires the formula G to be true at p. Like Java assertions, Jahob assertions identify conditions that should be true at a given program point. But where Java assertions are dynamically checked for only the current execution, Jahob assertions are statically checked to hold for all executions. In particular, Jahob assertions produce proof obligations that Jahob statically verifies to guarantee that G will be true in all program executions that satisfy the precondition of the method.

An assert G statement can also optionally contain a clause "from $l_1, \ldots, l_n$" to identify the facts from which G should follow. The identifiers l_i can refer to the labels of facts introduced by previous assume statements and proof commands, preconditions, named invariants, conditions encoding a path in the program, or parts of formulas explicitly labeled using the comment keyword. Jahob then passes this information to the automated provers through the generated verification condition formulas, by producing formulas that include only the specified facts. In this way, the from clause provides a means for the developer to guide the provers in establishing the desired proof. This is a novel aspect of Jahob's assertion mechanism not found in other program verification systems. By allowing the developer to control the facts

that Jahob passes to the provers, the from clause is, in effect, controlling the proof search space that the provers have to traverse. The addition of a from clause alone is often sufficient to limit the proof search space to enable a prover to prove a formula on which it previously failed. The from clause is particularly helpful for guiding first-order and SMT provers to an appropriate set of facts to use when the number and complexity of the invariants become large [27].

Assume

An assume G statement is dual to the assert statement. Whereas an assert requires Jahob to demonstrate that G holds, an assume statement allows Jahob to assume that G is true at a given program point. The developer-supplied use of assume statements may violate soundness and causes Jahob to emit a warning. The intended use of assume is debugging, because it allows Jahob to verify a method under the desired restricted conditions. For example, an assume false annotation at the beginning of a branch of an if statement means that Jahob will effectively skip the verification of that branch. More generally, assume statements allow the developer to focus the verification on a particular scenario of interest (e.g. a particular aliasing condition) and therefore understand better why a proof attempt is failing.

Proof Commands

Jahob's proof language allows the developer to guide the efforts of the automated provers in the event that the provers are unable to complete the verification automatically. A proof command at program point p in the body of a method directs Jahob to prove that a given property is true at p, and to use the proved lemma in subsequent verification. Proof commands may also inform Jahob as to how to prove the desired property. Chapter 4 describes the details of Jahob's proof language.

Other Annotations

To designate an object as being private to a class, the corresponding Java new statement is annotated with the hidden keyword. The purpose of such designation is for enabling more accurate contracts for public methods that modify private arrays (see Section 3.2.2). Jahob also supports the claimedby annotation for designating classes and fields as accessible only by a specified class. These annotations support access constraints similar to static nested classes (in the case of claimed classes), but also enable finer grain access control in the case of claimed fields.

3.2 Generating Verification Conditions

Jahob produces verification conditions by simplifying the Java code and transforming it into extended guarded commands (Figure 3-4), then desugaring extended guarded

commands into simple guarded commands (Figure 3-6), and finally generating verification conditions from simple guarded commands in the standard way using weakest liberal preconditions (Figure 3-7).

$$
\begin{array}{rcl}
c & ::= & p \\
& | & \mathsf{assume}\ l{:}\ F \\
& | & \mathsf{assert}\ l{:}\ F\ \mathsf{from}\ \vec{h} \\
& | & \mathsf{havoc}\ \vec{x}\ \mathsf{suchThat}\ l{:}\ F \\
& | & \mathsf{skip} \mid c_1 \,[]\, c_2 \mid c_1 \,;\, c_2 \\
& | & x := F \\
& | & \mathsf{if}(F)\ c_1\ \mathsf{else}\ c_2 \\
& | & \mathsf{loop\ inv}(I)\ c_1\ \mathsf{while}(F)\ c_2
\end{array}
$$

Figure 3-4: Extended Guarded Commands

3.2.1 Representation of Program Memory

In Jahob, the state of a program is given by a finite number of concrete and specification variables. The type of a specification variable appears in its declaration. Jahob maps the types of concrete Java variables as follows. Static reference variables become variables of type obj, where obj is the type of all object identifiers. An instance variable f in a class declaration class C {D f} becomes a function $f :: \mathsf{obj} \Rightarrow \mathsf{obj}$ mapping object identifiers to object identifiers. The Java expression x.f becomes fx, that is, the function f applied to x. Jahob represents Java class information using a set of objects for each class. For example, Jahob generates the axiom $\forall x.x \in C \rightarrow fx \in D$ for the above field f. Note that the function f is total. When x is null or of a class that does not include the field f, $fx = \mathsf{null}$. (Jahob correctly checks for the absence of null dereferences by creating an explicit assertion before each dereference.) Jahob represents object-valued arrays as a function of type $\mathsf{obj} \Rightarrow \mathsf{int} \Rightarrow \mathsf{obj}$, which accepts an array and an index and returns the value of the array at the index. Jahob also introduces a function of type $\mathsf{obj} \Rightarrow \mathsf{int}$ that indicates the array size, and uses it to generate array bounds check assertions. The type int represents the integer type, which Jahob models as the set of unbounded mathematical integers.

To account for aliasing between arrays, Jahob uses a special arrayState variable to represent the array component of the program state at method entry. Array reads take an array state, an array, and an index, and produce the corresponding value, while array writes take an array state, array, index, value, and produce the updated array state.

Jahob also uses a special alloc set to represent the set of all allocated objects. The set alloc is updated automatically by the system to account for allocations, and may be referred to, but not modified, by the developer. Unlike the conventional notion of allocation, alloc is also monotonic. Conceptually, once an object is allocated, it continues to have been allocated, and is never removed from the set. An object that is

$$
\begin{array}{rcl}
p & ::= & p_1 \,;\, p_2 \\
& | & \mathsf{note}\ l{:}F\ \mathsf{from}\ \vec{h} \\
& | & \mathsf{localize\ in}\ (p \,;\, \mathsf{note}\ l{:}F) \\
& | & \mathsf{mp}\ l{:}(F \rightarrow G) \\
& | & \mathsf{assuming}\ l_F{:}F\ \mathsf{in}\ (p \,;\, \mathsf{note}\ l_G{:}G) \\
& | & \mathsf{cases}\ \vec{F}\ \mathsf{for}\ l{:}G \\
& | & \mathsf{showCase}\ i\ \mathsf{of}\ l{:}F_1 \vee \ldots \vee F_n \\
& | & \mathsf{byContradiction}\ l{:}F\ \mathsf{in}\ p \\
& | & \mathsf{contradiction}\ l{:}F \\
& | & \mathsf{instantiate}\ l{:}\forall \vec{x}.F\ \mathsf{with}\ \vec{t} \\
& | & \mathsf{witness}\ \vec{t}\ \mathsf{for}\ l{:}\exists \vec{x}.F \\
& | & \mathsf{pickWitness}\ \vec{x}\ \mathsf{for}\ l_F{:}F\ \mathsf{in}\ (p \,;\, \mathsf{note}\ l_G{:}G) \\
& | & \mathsf{pickAny}\ \vec{x}\ \mathsf{in}\ (p \,;\, \mathsf{note}\ l{:}F) \\
& | & \mathsf{induct}\ l{:}F\ \mathsf{over}\ n\ \mathsf{in}\ p
\end{array}
$$

Figure 3-5: Integrated Proof Language Commands

$$
\begin{array}{rcl}
c & ::= & \mathsf{assume}\ l{:}\,F \\
& | & \mathsf{assert}\ l{:}\,F\ \mathsf{from}\ \vec{h} \\
& | & \mathsf{havoc}\ \vec{x} \\
& | & \mathsf{skip} \mid c_1 \,[]\, c_2 \mid c_1 \,;\, c_2
\end{array}
$$

Figure 3-6: Simple Guarded Commands

$$
\begin{array}{lcl}
\mathsf{wlp}((\mathsf{assume}\ l{:}\,F), G) & = & F^{[l]} \rightarrow G \\
\mathsf{wlp}((\mathsf{assert}\ l{:}\,F\ \mathsf{from}\ \vec{h}), G) & = & F^{[l;\vec{h}]} \wedge G \\
\mathsf{wlp}((\mathsf{havoc}\ \vec{x}), G) & = & \forall \vec{x}.\, G \\
\mathsf{wlp}((\mathsf{skip}), G) & = & G \\
\mathsf{wlp}((c_1 \,[]\, c_2), G) & = & \mathsf{wlp}(c_1, G) \wedge \mathsf{wlp}(c_2, G) \\
\mathsf{wlp}((c_1 \,;\, c_2), G) & = & \mathsf{wlp}(c_1, \mathsf{wlp}(c_2, G))
\end{array}
$$

Figure 3-7: Weakest Preconditions for Simple Guarded Commands

not in alloc is *isolated*—i.e. all of its fields are null and there are no references to it. The object allocation primitive new returns an arbitrary new object that was not in the set of allocated objects before the allocation. For this approach to work, the specification must require that every object that the program manipulates be an element of the set of allocated objects. This fact is ensured by the operational semantics of the program, but must be stated and inductively established for the verification to succeed. Jahob automatically inserts the appropriate clauses in method contracts to ensure that this property holds.

3.2.2 Class-Level Encapsulation

Jahob supports enforcement of class-level encapsulation by means of an abstract, per-class, set of hidden objects—objects that are private to the class. The formula expression C.hidden refers to a system-defined ghost variable that is automatically defined for every class C when using Jahob with class-level encapsulation turned on. Jahob inserts the appropriate checks to ensure that objects in the set do not escape the class. Jahob also emits the specification assignment statements that modify membership in the set, in accordance with developer annotations that identify hidden objects. As with alloc, hidden is monotonic—objects may be added to but not removed from hidden—and may not be modified except by the system.

In general, the purpose of class-level encapsulation is to enable more accurate method contracts for public methods that modify private arrays. Without the concept of a hidden set, or a mechanism that serves a similar purpose, contracts for such methods would need to list arrayState in the modifies clause, indicating that any array may be modified, as contracts of public methods may not refer to private fields. When using class-level encapsulation, the frame condition for arrays allows for the modification of arrays in the hidden set, making it possible to soundly omit the mention of arrayState in the modifies clause.

3.2.3 From Java to Guarded Commands

Jahob's transformation of the annotated Java program into guarded commands resembles a compilation process, and is described formally in Figures 3-8, 3-9, and 3-10. Jahob simplifies executable statements into three-address form to make the evaluation order in expressions explicit. It also inserts assertions that check for null dereferences, array bounds violations, and type cast errors. It converts field and array assignments into assignments of global variables whose right-hand side contains function update expressions. Having taken side effects into account, it transforms Java expressions into mathematical expressions in higher-order logic.

Transformations for hidden Objects

Jahob automatically inserts checks to ensure that hidden objects do not escape the class. A hidden object may not be passed as a parameter, returned from a method, assigned to a field of an object that is not hidden, or assigned to an array that

$$
\begin{array}{rcl}
[\![v_1 = v_2]\!] & = & v_1 := v_2 \\
[\![v = C.f]\!] & = & v := C.f \\
[\![C.f = e]\!] & = & [\![v_{\mathsf{fresh}} = e]\!] \,;\, C.f := v_{\mathsf{fresh}} \\
[\![v_1 = v_2.f]\!] & = & \mathsf{assert}\ v_2 \neq \mathsf{null} \,;\, v_1 := f\ v_2 \\
[\![v.f = e]\!] & = & [\![v_{\mathsf{fresh}} = e]\!] \,;\, \mathsf{assert}\ v \neq \mathsf{null} \,;\, f := f(v := v_{\mathsf{fresh}}) \\
[\![v_1 = v_2[e]]\!] & = & [\![v_{\mathsf{fresh}} = e]\!] \,; \\
 & & \mathsf{assert}\ (v_2 \neq \mathsf{null} \wedge 0 \leq v_{\mathsf{fresh}} \wedge v_{\mathsf{fresh}} < \mathsf{length}\ v_2) \,; \\
 & & v_1 := \mathsf{arrayState}\ v_2\ v_{\mathsf{fresh}} \\
[\![v[e_1] = e_2]\!] & = & [\![v_{\mathsf{fresh},1} = e_1 \,;\, v_{\mathsf{fresh},2} = e_2]\!] \,; \\
 & & \mathsf{assert}\ (v \neq \mathsf{null} \wedge 0 \leq v_{\mathsf{fresh},1} \wedge v_{\mathsf{fresh},1} < \mathsf{length}\ v) \,; \\
 & & \mathsf{arrayState} := (\mathsf{arrayState}\ v)(v_{\mathsf{fresh},1} := v_{\mathsf{fresh},2}) \\
[\![v = \mathsf{new}\ C(e_1, \ldots, e_n)]\!] & = & \mathsf{havoc}\ v \,; \\
 & & \mathsf{assume}\ (v \neq \mathsf{null} \wedge v \notin \mathsf{alloc} \wedge v \in C \wedge \mathit{lonely}(v)) \,; \\
 & & \mathsf{alloc} := \mathsf{alloc} \cup \{v\} \,; [\![C(v, e_1, \ldots, e_n)]\!] \\
[\![v = \mathsf{new}\ C[e]]\!] & = & [\![v_{\mathsf{fresh}} = e]\!] \,; \mathsf{havoc}\ v \,; \\
 & & \mathsf{assume}\ (v \neq \mathsf{null} \wedge v \notin \mathsf{alloc} \wedge v \in \mathsf{Array}) \,; \\
 & & \mathsf{assume}\ (\mathsf{length}\ v = v_{\mathsf{fresh}} \wedge \mathit{lonely}(v)) \,; \\
 & & \mathsf{alloc} := \mathsf{alloc} \cup \{v\} \\
[\![v = (e_1\ op\ e_2)]\!] & = & [\![v_{\mathsf{fresh},1} = e_1 \,;\, v_{\mathsf{fresh},2} = e_2]\!] \,; \\
 & & \mathsf{assert}\ v_{\mathsf{fresh},2} \neq 0 \,;\ (\text{if } op \text{ is } /) \\
 & & v := v_{\mathsf{fresh},1}\ \mathit{binop}(op)\ v_{\mathsf{fresh},2} \\
[\![v = op\ e]\!] & = & [\![v_{\mathsf{fresh}} = e]\!] \,; v := \mathit{unop}(op)\ v_{\mathsf{fresh}} \\
[\![\text{while /*: inv “}F\text{” */ } (e)\ \{s\}]\!] & = & \mathsf{loop\ inv}(F)\ [\![v_{\mathsf{fresh}} = e]\!]\ \mathsf{while}(v_{\mathsf{fresh}})\ [\![s]\!] \\
[\![\text{if } (e)\ \{s_1\}\ \text{else}\ \{s_2\}]\!] & = & [\![v_{\mathsf{fresh}} = e]\!] \,; \mathsf{if}(v_{\mathsf{fresh}})\ [\![s_1]\!]\ \text{else}\ [\![s_2]\!] \\
[\![\text{return } v]\!] & = & \mathsf{result} := v \\
 & & \\
\text{where } \mathit{lonely}(v) & = & (\forall x.\ f_1\ x \neq v) \wedge \ldots \wedge (\forall x.\ f_n\ x \neq v) \wedge \\
\text{(for instance fields } f_1, \ldots, f_n & & (g_1 \neq v) \wedge \ldots \wedge (g_m \neq v) \wedge \\
\text{and static fields } g_1, \ldots, g_m & & (\forall y\ i.\ \mathsf{arrayState}\ y\ i \neq v) \wedge \\
\text{of reference type)} & & (f_1\ v = \mathsf{null}) \wedge \ldots \wedge (f_n\ v = \mathsf{null}) \wedge \\
 & & (\forall j.\ \mathsf{arrayState}\ v\ j = \mathsf{null}) \\
\mathit{binop}(!=) & = & \neq \\
\mathit{binop}(==) & = & = \\
\mathit{binop}(\&\&) & = & \wedge \\
\mathit{binop}(||) & = & \vee \\
\mathit{binop}(op) & = & op \text{ (for all other binary operators)} \\
\mathit{unop}(!) & = & \neg \\
\text{and } \mathit{unop}(-) & = & -
\end{array}
$$

Figure 3-8: Translating Java Statements into Extended Guarded Commands (continued in Figure 3-9)

$$
\begin{array}{rcl}
[\![v = m(e_1, \ldots, e_n)]\!] & = & [\![v_{\mathsf{fresh},1} = e_1 \,;\ldots;\, v_{\mathsf{fresh},n} = e_n]\!] \,; \\
\text{(for old } w_1, \ldots, \text{old } w_j & & \mathsf{assert}\ v_{\mathsf{fresh},1} \neq \mathsf{null}\,; \text{ (if } m \text{ is an instance method)} \\
\text{in } ensures_m) & & \mathsf{assert}\ requires'_m[p_i = v_{\mathsf{fresh},i}]\,; \\
& & w'_1 := w_1 \,;\ldots;\, w'_j := w_j \,; \\
& & \mathsf{havoc}\ modifies'_m\,;\ \mathsf{havoc}\ v\,; \\
& & \mathsf{assume}\ ensures'_m[p_i := v_{\mathsf{fresh},i}; \mathsf{result} := v; \mathsf{old}\ w_i := w'_i] \\
\\
\text{where } requires'_m & = & requires_m \wedge public\_invs(m) \\
public\_invs(m) & = & inv_1 \wedge \ldots \wedge inv_k \\
& & \text{(where } m \text{ is a method in class } C \text{ and } inv_1, \ldots, inv_k \text{ are} \\
& & \text{invariants of } C \text{ that depend on public fields and} \\
& & \text{specification variables of other classes)} \\
modifies'_m & = & f_1, \ldots, f_p, \mathsf{alloc}, rep\_vars(m) \\
& & \text{(where } f_1, \ldots, f_p \text{ are fields in } modifies_m) \\
rep\_vars(m) & = & g_1, \ldots, g_r \text{ (if } m \text{ is re-entrant)} \\
& & \text{(where } m \text{ is in class } C, \text{ and } g_1, \ldots, g_r \text{ are private fields} \\
& & \text{of } C \text{ or public fields claimed by } C) \\
rep\_vars(m) & = & [] \text{ (otherwise)} \\
ensures'_m & = & ensures_m \wedge public\_invs(m) \wedge \mathsf{old}\ \mathsf{alloc} \subseteq \mathsf{alloc} \wedge \\
& & frame(f_1) \wedge \ldots \wedge frame(f_r) \wedge array\_frame(x_1, \ldots, x_s) \\
& & \text{(where } f_1, \ldots, f_r \text{ are fields in } modifies_m \text{ in the form of} \\
& & x..f \text{ for some } x)) \\
frame(f) & = & \forall x.\ x \in \mathsf{old}\ \mathsf{alloc} \wedge x \in C \wedge hidden(x) \wedge x \neq \mathsf{old}\ y_1 \wedge \ldots \wedge \\
& & x \neq \mathsf{old}\ y_t \rightarrow f\ x = (\mathsf{old}\ f)\ x \\
& & \text{(where } f \text{ is a field in } C \text{ and } y_1..f, \ldots, y_t..f \text{ in } modifies_m) \\
hidden(x) & = & \mathsf{true} \text{ (if class-level encapsulation is enabled)} \\
hidden(x) & = & x \notin \mathsf{hidden} \text{ (otherwise)} \\
array\_frame() & = & \mathsf{true} \\
array\_frame(x_1, \ldots, x_s) & = & \forall y\ i.\ 0 \leq i \wedge i < \mathsf{length}\ y \wedge y \in \mathsf{Array} \wedge hidden(y) \wedge \\
& & y \neq \mathsf{old}\ x_1 \wedge \ldots \wedge y \neq \mathsf{old}\ x_s \wedge alloc(y) \rightarrow \\
& & \mathsf{arrayState}\ y\ i = (\mathsf{old}\ \mathsf{arrayState})\ y\ i \\
& & \text{(where } x_1..\mathsf{arrayState}, \ldots, x_s..\mathsf{arrayState} \text{ in } modifies_m) \\
alloc(y) & = & y \in \mathsf{old}\ \mathsf{alloc} \text{ (if } \mathsf{new}..\mathsf{arrayState} \text{ in } modifies_m) \\
alloc(y) & = & \mathsf{true} \text{ (otherwise)}
\end{array}
$$

Figure 3-9: Translating Java Statements into Extended Guarded Commands (continued from Figure 3-8)

$$
\begin{array}{rcl}
[\![\text{“}g\text{”} := \text{“}F\text{”}]\!] & = & g := F \\
[\![\text{“}v..g\text{”} := \text{“}F\text{”}]\!] & = & g := g(v := F) \\
[\![\mathsf{havoc}\ v\ \mathsf{suchThat}\ l\text{: “}F\text{”}]\!] & = & \mathsf{havoc}\ v\ \mathsf{suchThat}\ l\text{:}\ F
\end{array}
$$

Figure 3-10: Translating Specification Assignment Statements and Non-Deterministic Change into Extended Guarded Commands

is not hidden. For Java new statements annotated with the hidden keyword, Jahob automatically inserts a specification assignment statement adding the allocated object to the hidden set.

Receiver Parameters in Specifications

Java makes most uses of the receiver parameter this implicit, with the compiler using scoping rules to resolve such references. Jahob applies similar rules to disambiguate occurrences of variables in specifications. When a field f occurs in an expression that is not immediately of the form x..f and when f is not qualified with a class name, Jahob converts the occurrence of f into this..f. Jahob also transforms each definition x = f of a non-static specification variable x into the definition x = λthis.f. If, after the transformation, a class invariant *Inv* in class C contains an occurrence of this, Jahob transforms the invariant into $\forall \mathsf{this.this} \in C \wedge \mathsf{this} \in \mathsf{alloc} \rightarrow Inv$. An invariant stated for a given object is therefore implicitly interpreted as being required for all allocated objects of the class, and becomes a global invariant. This mechanism enables Jahob developers to not only concisely state invariants on a per-object basis but also to use global invariants that state relationships between different instances of the class.

3.2.4 From Extended to Simple Guarded Commands

The main internal representation of Jahob is the extended guarded command language. It contains guarded command statements, simple control structures, and proof statements. Figure 3-4 presents the syntax of the extended guarded command language. We next describe how Jahob transforms such guarded commands into the simple guarded command language (for which verification condition generation is standard as presented in Figure 3-7). For the moment, we omit discussion of proof commands p, which we address in Chapter 4.

3.2.5 Translating Extended Guarded Commands

Figure 3-11 describes the translation of extended guarded commands (modulo proof commands) into simple guarded commands. We translate assignments into havoc followed by an equality constraint, which reduces all state changes to havoc statements. Conditional statements become non-deterministic choice with assume statements, as in control-flow graph representations. The Jahob encoding of loops with loop invariants is analogous to the sound version of the encoding in ESC/Java [57].

3.2.6 Accounting for Variable Dependencies

The semantics of extended guarded commands assumes a set D of specification variable definitions (v, D_v) where v is a variable and D_v is a term representing the definition of v in terms of other variables. For a list of variables $\vec{u}$, we write deps($\vec{u}$) for the set of all variables that depend on any of the variables in $\vec{u}$, that is, variables whose value may change if one of the variables $\vec{u}$ changes. To define this set precisely, let

$$
\begin{array}{rcl}
[\![x := F]\!] & = & \mathsf{havoc}\ v\,;\mathsf{assume}\ (v = F)\,; \\
\text{(where } v \text{ is a fresh variable)} & & \mathsf{havoc}\ x\,;\mathsf{assume}\ (x = v) \\
[\![\mathsf{if}(F)\ c_1\ \mathsf{else}\ c_2]\!] & = & (\mathsf{assume}\ F\,;\ [\![c_1]\!])\ [] \\
& & (\mathsf{assume}\ \neg F\,;\ [\![c_2]\!]) \\
[\![\mathsf{loop\ inv}(I)\ c_1\ \mathsf{while}(F)\ c_2]\!] & = & \mathsf{assert}\ I\,;\mathsf{havoc}\ \vec{r}\,;\mathsf{assume}\ I\,; \\
\text{(where } \vec{r} = \mathsf{mod}(c_1;c_2) \text{ denotes} & & [\![c_1]\!]\,; \\
\text{variables modified in } c_1,\ c_2) & & (\mathsf{assume}\ (\neg F)\ [](\mathsf{assume}\ F\,; \\
& & \qquad [\![c_2]\!]\,;\mathsf{assert}\ I\,; \\
& & \qquad \mathsf{assume\ false})) \\
[\![\mathsf{havoc}\ \vec{x}\ \mathsf{suchThat}\ F]\!] & = & \mathsf{assert}\ \exists \vec{x}.F\,; \\
& & \mathsf{havoc}\ \vec{x}\,;\mathsf{assume}\ F
\end{array}
$$

Figure 3-11: Translating Extended Guarded Commands

$\mathsf{FV}(G)$ denote all free variables in G, let the dependence relation be $\rho = \{(v_1, v_2) \mid (v_2, D_2) \in D \wedge v_1 \in \mathsf{FV}(D_2)\}$ and let ρ^* denote the transitive closure of ρ. Then $\mathsf{deps}(u_1, \ldots, u_n) = \cup_{i=1}^{n}\{v \mid (u_i, v) \in \rho^*\}$. We write $\mathsf{defs}(\vec{u})$ for the set of constraints expressing these dependencies, with $\mathsf{defs}(\vec{u}) = \bigwedge\{v = D_v \mid v \in \mathsf{deps}(\vec{u}) \wedge (v, D_v) \in D\}$. To correctly take dependencies into account during verification condition generation, it suffices to treat each command of the form $\mathsf{havoc}\ \vec{x}$ in Figure 3-11 as the command $(\mathsf{havoc}\ (\vec{x}, \mathsf{deps}(\vec{x}))\,;\mathsf{assume}\ \mathsf{defs}(\vec{x}))$. The same applies to the translation of proof commands in Figure 4-1.

Eliminating Unnecessary Assumptions

To simplify the generated verification conditions, some of the internally generated assume statements indicate a variable that the statement is intended to constrain. For example, an assumption generated from a variable definition $v = D_f$ is meant to constrain the variable v, as are assumptions of the form $v \in C$ where C is a set of objects of class C. Ignoring an assumption is always sound, and Jahob does so whenever the postcondition does not contain a variable that the assumption is intended to constrain. Moreover, in certain cases Jahob reorders consecutive assume statements to increase the number of assumptions that it can omit.

3.3 Proving Verification Conditions

Jahob generates a proof obligation for each method it verifies. It also generates a proof obligation to ensure the consistency of the class invariants by verifying that these invariants hold in the initial state of the program. These verification conditions are expressed in a subset of the Isabelle/HOL notation. We next discuss how Jahob proves such verification conditions.

$$\begin{array}{rcl} \vec{A} \rightarrow G_1 \wedge G_2 & \leadsto & \vec{A} \rightarrow G_1,\ \vec{A} \rightarrow G_2 \\ \vec{A} \rightarrow (\vec{B} \rightarrow G^{[p]})^{[q]} & \leadsto & (\vec{A} \wedge \vec{B}^{[q]}) \rightarrow G^{[pq]} \\ \vec{A} \rightarrow \forall x.G & \leadsto & \vec{A} \rightarrow G[x := x_{\mathsf{fresh}}] \end{array}$$

Figure 3-12: Splitting Rules for Converting a Formula into an Implication List ($F^{[c]}$ denotes a formula F annotated with a string c)

3.3.1 Splitting

Jahob follows the standard rules of weakest liberal preconditions shown in Figure 3-7 to generate verification conditions. Verification conditions generated using these rules can typically be represented as a conjunction of a large number of conjuncts. Figure 3-12 describes Jahob's splitting process, which produces a list of implications whose conjunction is equivalent to the original formula. The individual implications correspond to different paths in the method, as well as different conjuncts of assert statements, operation preconditions, invariants, postconditions, and preconditions of invoked methods.

The splitting rules in Jahob preserve formula annotations, which are used for assumption selection and in error messages to indicate why a verification failed. Because Jahob splits only the goal of an implication, the number of generated implications is polynomial in the size of the original verification condition (the verification condition itself can be exponential in the size of the method). During splitting Jahob eliminates simple syntactically valid implications, such as those whose goal occurs as one of the assumptions, using an internal syntactic prover.

3.3.2 Using Multiple Provers

Figure 3-13 shows the Jahob system diagram including the provers to which Jahob interfaces. A typical data structure operation generates a verification condition that splitting separates into a few hundred implications, each of which is a candidate for any of the provers in Figure 3-13. Each implication generated from a verification condition must be valid for the data structure operation to be correct. Each proof can be performed entirely independently.

To prove an implication, Jahob may attempt to use any of the available provers. In practice, a Jahob user specifies, for a given verification task, a sequence of provers and their parameters on the command line. Jahob tries the provers in sequence, so the user lists the provers starting from the ones that are most likely to succeed or, if possible, fail quickly when they do not succeed. Often different provers are appropriate for different proof obligations in the same method. Because the generated proof obligations are entirely independent, Jahob also provides a facility to spawn provers in parallel, which can significantly reduce the overall proof time on multi-core machines.

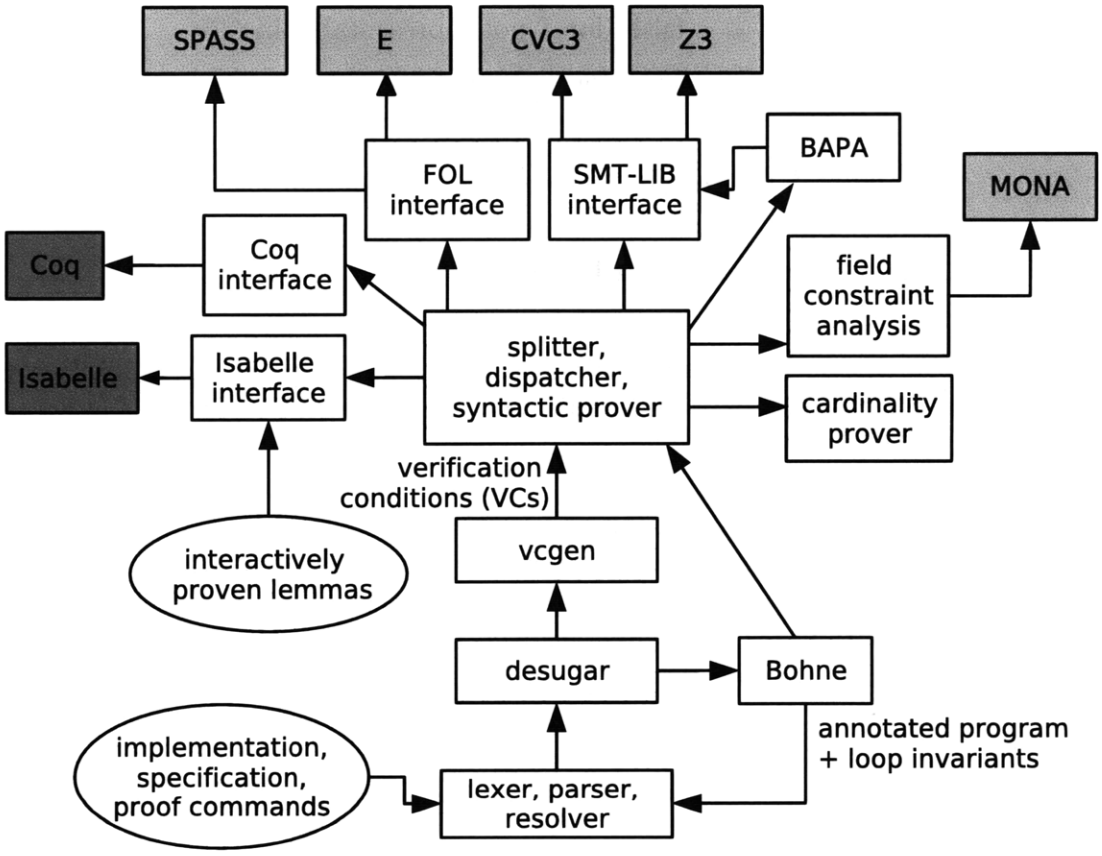


Figure 3-13: Jahob System Diagram

$$
\begin{aligned}
\alpha &: \{0,1\} \times F \rightarrow C \\
\alpha^p(f_1 \wedge f_2) &\equiv \alpha^p(f_1) \wedge \alpha^p(f_2) \\
\alpha^p(f_1 \vee f_2) &\equiv \alpha^p(f_1) \vee \alpha^p(f_2) \\
\alpha^p(\neg f) &\equiv \neg\alpha^{\overline{p}}(f) \\
\alpha^p(\forall x.f) &\equiv \forall x.\alpha^p(f) \\
\alpha^p(\exists x.f) &\equiv \exists x.\alpha^p(f) \\
\alpha^p(f) &\equiv e, \text{ for } f \text{ directly representable in } C \text{ as } e \\
\alpha^0(f) &\equiv \mathsf{false}, \text{ for } f \text{ not representable in } C \\
\alpha^1(f) &\equiv \mathsf{true}, \text{ for } f \text{ not representable in } C
\end{aligned}
$$

Figure 3-14: General Formula Approximation Scheme

3.3.3 Formula Approximation

Efficient provers are often specialized for a particular class of formulas. One of the distinguishing characteristics of Jahob is its ability to integrate such specialized provers into a system that uses an expressive fragment of higher-order logic. This integration is based on the concept of *formula approximation*, which maps an arbitrary formula into a semantically stronger but simpler formula in an appropriate subset of higher-order logic. Because the resulting formula is stronger, the approach is sound.

Figure 3-14 presents the general scheme for formula approximation: for atomic formulas representable in the target logic subset, the approximation produces the appropriate translation; for logical operations it proceeds recursively; for unsupported atomic formulas it produces true or false depending on the polarity of the formula. To improve the precision of this recursive approximation step, Jahob first applies rewrite rules that substitute the definitions of values, perform beta reduction, and flatten expressions. The details of rewriting and approximation depend on the individual prover interface.

3.4 Provers Deployed in Jahob

Jahob supports a number of types of provers, both internal and external. Here we describe the provers that are deployed in Jahob.

3.4.1 Syntactic Prover

Before invoking any other internal or external provers, Jahob first tests whether a formula is trivially valid using its internal syntactic prover. Specifically, it checks for the presence of appropriately placed propositional constants false and true. It also checks whether or not the conclusion of an implication appears in the assumption

(modulo simple syntactic transformations that preserve validity). In practice these techniques discharge many verification condition conjuncts. One source of such conjuncts is checks, such as null dereference checks, which occur (implicitly) many times in the source code. Another source is sequences of method calls. Specifically, when class invariants that hold after one method call need to be shown to hold for subsequent calls. Sequences of proof commands may also produce proof obligations that are discharged by earlier commands. For complex formulas the syntactic prover is very useful because more sophisticated provers often perform transformations that destroy the structure of the formula, converting it into a form for which the proof attempt fails.

3.4.2 First-Order Provers

Decades of research into first-order theorem proving by resolution have produced carefully engineered systems capable of proving non-trivial first-order formulas [152, 156, 149]. Jahob leverages this development by translating higher-order logic into first-order logic [27]. This translation is very effective for formulas without transitive closure and arithmetic. Such formulas may contain set expressions, but those expressions are typically quantifier-free, which enables their translation into quantified first-order formulas. Using ghost variables and recursive axioms, we are also able to use first-order provers to prove strong properties about reachability in data structures [27]. Our translation uses an incomplete set of axioms for ordering and addition to provide partial support for linear arithmetic. We found this axiomatization effective for reasoning about data structures such as priority queues and binary search trees, which depend on arithmetic properties.

3.4.3 SMT Provers

Provers based on Nelson-Oppen combination of decision procedures enhanced with quantifier instantiation have been among the core technologies of past verification systems [119]. Jahob incorporates state-of-the art solvers in this family using the SMT-LIB standard format [132]. The approximation for this format is similar to the approximation for first-order provers, but uses the SMT-LIB representation of linear arithmetic. We use two SMT provers in Jahob: CVC3 [60] and Z3 [116, 117].

3.4.4 MONA

MONA is a decision procedure for monadic second-order logic over strings and trees [67]. We use it primarily for the verification of shape properties. Its expressive power stems from its ability to quantify over sets of objects. Quantification over sets can in turn encode transitive closure, which is extremely useful for reasoning about recursive data structures. Jahob contains a flexible interface that enables the use of MONA even for some non-tree data structures [159]. When proving an implication $A_1, \ldots, A_n \rightarrow G$, this interface identifies assumptions of the form $\mathsf{tree}[f_1, \ldots, f_n]$, then interprets the formula assuming that $f_1, \ldots, f_n$ form the tree

backbone of the data structure. Furthermore, it identifies assumptions A_i of the form $\forall xy.f(x)=y \rightarrow H(x,y)$ (for $f \notin \{f_1,\ldots,f_n\}$) and soundly approximates a goal of the form $G(f(t))$ with the stronger goal $\forall u.H(t,u) \rightarrow G(u)$. This enables the approximation to maintain information about non-tree fields and provides certain completeness guarantees [159, Theorems 2 and 3].

3.4.5 BAPA

Jahob also implements a decision procedure for sets with symbolic cardinality bounds [87, 84]. This decision procedure can prove a class of verification conditions that use set algebra, symbolic cardinality constraints, and linear arithmetic (i.e., quantifier-free Presburger arithmetic). Such verification conditions arise when checking invariants on the size of allocated structures and other examples such as tracking the number of objects that a method allocates [87]. Previous theorem provers have limited effectiveness for such formulas because set algebra and linear arithmetic interact in non-trivial ways through the cardinality operator.

3.4.6 Simple Cardinality Prover

Jahob's simple cardinality prover is a fast, efficient solver that uses simple syntactic rules to discharge verification conditions that involve the cardinality operator for finite sets, according to the rules in Figure 3-15. These rules correspond to verification conditions that commonly arise in the verification of data structure implementations. While the BAPA decision procedure is theoretically capable of proving many of these formulas, the current implementation of BAPA supports only sets of obj type. Since sets of tuples often arise in data structures whose abstract state involve relations (e.g. hash tables, association lists, priority queues, etc.), the simple cardinality prover is critical to the successful verification of such data structures.

3.4.7 Isabelle and Coq

Jahob provides interfaces to interactive theorem provers Isabelle [122] and Coq [24]. Jahob can invoke Isabelle automatically on a given proof obligation using the general-purpose theorem proving tactic in Isabelle. In some cases (e.g., for relatively small proof obligations that involve complex set expressions) this approach succeeds even when other approaches fail. In general, Isabelle requires interaction, so the user can prove the implication interactively and save it into a file. Jahob loads this file in future verification attempts and treats such proved lemmas as true. It determines whether a given proof obligation is a proved lemma using a matching algorithm that is able to match not only syntactically identical formulas but also certain classes of semantically equivalent and stronger formulas by taking into account renamed variables, and reordered and extraneous assumptions. For example, if the user has saved as proved a formula of the form $f \rightarrow h$, Jahob is able to match the formula $f \wedge g \rightarrow h$ and determine that it is proved on the basis of the interactive proof for the stronger formula.

$$\frac{X = Y \cup \{e\} \qquad e \notin Y \qquad \mathsf{card}(Y) = c}{\mathsf{card}(X) = c + 1}$$

$$\frac{X = Y \cup \{e\} \qquad e \notin Y \qquad \mathsf{card}(X) = c + 1}{\mathsf{card}(Y) = c}$$

$$\frac{X = Y \cup \{e\} \quad \mathsf{card}(X) = c + 1 \qquad \mathsf{card}(Y) = c}{e \notin Y}$$

$$\frac{X = Y \cup \{e\} \qquad e \notin Y \qquad \mathsf{card}(Y) = c - 1}{\mathsf{card}(X) = c}$$

$$\frac{X = Y \cup \{e\} \qquad e \notin Y \qquad \mathsf{card}(X) = c}{\mathsf{card}(Y) = c - 1}$$

$$\frac{X = Y \cup \{e\} \quad \mathsf{card}(X) = c \quad \mathsf{card}(Y) = c - 1}{e \notin Y}$$

$$\frac{X = Y - \{e\} \qquad e \in Y \qquad \mathsf{card}(Y) = c}{\mathsf{card}(X) = c - 1}$$

$$\frac{X = Y - \{e\} \qquad e \in Y \qquad \mathsf{card}(X) = c - 1}{\mathsf{card}(Y) = c}$$

$$\frac{X = Y - \{e\} \quad \mathsf{card}(X) = c - 1 \qquad \mathsf{card}(Y) = c}{e \in Y}$$

$$\frac{X = Y - \{e\} \qquad e \in Y \qquad \mathsf{card}(Y) = c + 1}{\mathsf{card}(X) = c}$$

$$\frac{X = Y - \{e\} \qquad e \in Y \qquad \mathsf{card}(X) = c}{\mathsf{card}(Y) = c + 1}$$

$$\frac{X = Y - \{e\} \quad \mathsf{card}(X) = c \quad \mathsf{card}(Y) = c + 1}{e \in Y}$$

$$\frac{\mathsf{card}(X) = 0}{X = \emptyset} \qquad \frac{X = \emptyset}{\mathsf{card}(X) = 0}$$

Figure 3-15: Simple Cardinality Prover Rules

3.5 Discussion

Certain aspects of Jahob's design make it particularly effective in the verification of full functional correctness. In particular, Jahob's specification language and use of integrated reasoning are critical to this application. On the other hand, the generality of other aspects of Jahob's design may be contributing to the difficulty of verifying programs in the system. We also discuss the nuances of verifying instantiable data structures, and note the limitations of the system.

3.5.1 Specification Language

As a specification language for Jahob, higher-order logic offers several benefits. First, its expressivity enables the specification of arbitrarily complex program properties. This is important for ensuring that all desired aspects of the program can be captured in the invariants, abstraction function, and method contracts of the verified program. Second, because higher-order logic is a standard logic with well-defined semantics, there are many existing tools for automated reasoning for both higher-order logic and its subsets. This property is essential for enabling our integrated reasoning approach.

One drawback, however, to the use of higher-order logic as a specification language, is the lack of concise counterparts for some common concepts in programming. For example, there is no map primitive in higher-order logic. Instead, maps are modeled using sets of pairs, with the additional constraint that each key is mapped to a unique value. Although the desired properties can still be expressed in higher-order logic, the resulting specifications may be more verbose (and consequently, more difficult to understand), than the equivalent specifications in a language with the desired primitives. Jahob also does not currently support the use of method calls in formulas. This ability can be useful in specifying the behavior of one method in terms of that of another. Instead, methods must be specified directly in terms of the change in abstract state. The resulting specifications may, again, be more verbose, and consequently, more difficult to understand, than the equivalent specifications in a language that supported method calls. It may, however, be possible to extend Jahob formulas with support for calls to pure methods (i.e. no side-effects) using assume-guarantee reasoning.[2] The system could also provide primitives for concepts such as maps, and internally translate these primitives to the corresponding higher-order logic representation.

In practice, we were able to use Jahob to specify all the properties needed for verifying the full functional correctness of our data structures. We also found the expressivity of higher-order logic useful for specifying shorthands using lambda expressions. This ability enabled more concise specifications, offsetting the lack of support for method calls and programming primitives in formulas. We used quantification over integers and objects extensively. While we did not use higher-order quantification in our examples, contexts exist where it may be useful. (For example, to specify that two representations are isomorphic, it may be necessary to quantify over functions.)

[2]Program verification systems that support the use of method calls in specifications typically only support pure methods, as the semantics for support of arbitrary method calls are unclear. Program analysis techniques exist for checking method purity [145].

Because we use higher-order logic as both a specification language and internal representation, we did not need to translate from one to the other. The generation of verification conditions was also straightforward. The availability of standard algorithms and tools enabled us to productively focus our efforts on the verification of more sophisticated properties.

3.5.2 Integrated Reasoning

With arbitrarily complex specifications come arbitrarily complex verification conditions. Our integrated reasoning approach makes it possible to apply highly-specialized automated reasoning techniques to these verification conditions. Splitting reduces these verification conditions into smaller formulas to enable the fine-grain application of different automated reasoning techniques. Approximation performs the sound translation necessary for the application of standard automated tools that support specialized logic subsets. Our approach enables the automated verification of properties that lie well beyond the ability of any single automated reasoning system to prove. This ability is essential, as the size and complexity of the verification conditions generated for full functional correctness verification make fully manual verification impractical.

We did not find any disadvantages to using the integrated reasoning approach in the verification of our data structure implementations. The availability of standard interfaces for first-order and SMT solvers made it easy to integrate new provers into the system. The integrated reasoning infrastructure also made it easy to incorporate new internal provers. Although there is some overlap in the abilities of provers in the same class, there are sufficient differences to make the use of multiple provers more effective than any single prover. However, there were still some properties that the system was unable to verify without guidance. These include properties that require a prover to search an unbounded search space in a bounded amount of time, or the use of multiple provers to prove. Our integrated proof language, which we describe in Chapter 4, addresses this problem by enabling developers to provide the system with the guidance needed to prove these verification conditions.

3.5.3 Overall Design

In general, Jahob was designed with the following philosophy. Whenever there was a choice between a more general mechanism, which enables the specification or verification of arbitrarily complex properties, and a more specific mechanism, which restricts what could be specified or verified, but made the annotated program easier for the system to process, we chose the more general mechanism. This philosophy can be seen in many aspects of the system, including the specification language, the semantics of the memory model, the supported provers, and the proof language. We did not want to restrict, *a priori*, the properties or programs that we could verify using the system. And in practice, we did not encounter any such properties or programs.

However, this generality can result in verification conditions that are more difficult to prove than what would be produced using more specific mechanisms. For

example, the use of a global arrayState provides the most general mechanism for specifying properties of arrays. It does not restrict the user's ability to specify arrays that may be aliased. However, this ability results in verification conditions that may require the consideration of many possible aliasing scenarios to prove, even though, in most cases, modified arrays are private to a particular object. Similarly, invariants asserted at method boundaries are universally quantified for all objects of the class. This policy allows for instance methods that may modify other instances of the class, but requires the re-establishment of the invariants for all instances. In most cases, however, instance methods invoked on one object do not affect the state of other instances. So although the generality of the system enabled the unrestricted specification and verification of properties necessary for full functional correctness, we may be able to produce verification conditions that are easier to prove, by improving the system's ability to detect and handle common cases.

3.5.4 Verifying Instantiable Data Structures

A Java class can implement a single static instance of a data structure or an instantiable data structure where multiple instances can be created. In many cases, the difference between the two implementations is no more than a handful of static keywords. But because of the possibility of aliasing between fields of different instances, instantiable data structures can be substantially more difficult to verify. Figures 3-16 and 3-17 illustrate the difference between instantiable and static data structures using excerpts from an instantiable version and a static version of the association list. Syntactically, the only difference between the two versions is the static modifier for the first field, the contents specification variable, and the add() method. The verification conditions that Jahob generates, however, are very different. In the static example, the invariant MapInv is simply as stated. But in the instantiable example, the reference to contents in the invariant actually refers to this..contents, which Jahob automatically resolves. Since the invariant must apply to all allocated instances of AssociationList, Jahob automatically adds the necessary quantification to produce the following invariant:

$$\forall \text{this.this} \in \text{AssociationList} \wedge \text{this} \in \text{alloc} \rightarrow$$
$$(\forall \text{k v0 v1.(k, v0)} \in \text{contents} \wedge \text{(k, v1)} \in \text{contents} \rightarrow \text{v0} = \text{v1})$$

This additional quantification can make the invariant more difficult to verify, since the original property must now be proved for multiple instances of the list, and not just one.

Similarly, the modifies clause of the add() method for the static association list refers to the static specification variable contents, while that of the instantiable association list refers to this..contents. To verify the modifies clause of the instantiable association list, Jahob must therefore ensure the following additional frame condition, which is not needed for the static version:

$$\forall \text{x.}\ \text{x} \in \text{AssociationList} \wedge \text{x} \in \text{old alloc} \wedge \text{x} \neq \text{this} \rightarrow \text{x..contents} = \text{old (x..contents)}$$

```
public class AssociationList {
    private Node first;
    /*:
      public specvar contents :: "(obj * obj) set";
      vardefs "contents == first..con";

      invariant MapInv:
      "∀k v0 v1. (k, v0) ∈ contents ∧ (k, v1) ∈ contents → v0 = v1";
      ...
    */
    public void add(Object k0, Object v0)
    /*: requires "k0 ≠ null ∧ v0 ≠ null ∧ ¬(∃v. (k0, v) ∈ contents)"
        modifies contents
        ensures "contents = old contents ∪ {(k0, v0)}" */
    { ... }
    ...
}
```

Figure 3-16: Instantiable Association List

```
public class AssociationList {
    private static Node first;
    /*:
      public static specvar contents :: "(obj * obj) set";
      vardefs "contents == first..con";

      invariant MapInv:
      "∀k v0 v1. (k, v0) ∈ contents ∧ (k, v1) ∈ contents → v0 = v1";
      ...
    */
    public static void add(Object k0, Object v0)
    /*: requires "k0 ≠ null ∧ v0 ≠ null ∧ ¬(∃v. (k0, v) ∈ contents)"
        modifies contents
        ensures "contents = old contents ∪ {(k0, v0)}" */
    { ... }
    ...
}
```

Figure 3-17: Static Association List

The correctness of the frame condition depends on the lack of sharing between the state of the currently modified association list and the state of other instances. For some instantiable data structures, the developer must write additional invariants, not needed for the static version of the same data structure, to express this lack of sharing. In some cases, the additional complexity may cause an instantiable data structure to require proof commands to verify, where the equivalent static data structure would not.

While our verified data structures do not make use of any special analyses for instantiable data structures, Jahob supports analyses for identifying data structure implementations where different instances do not share state. It is possible to extend Jahob to use these analyses to generate simpler verification conditions for these data structures, which could simplify the resulting verifications.

3.5.5 Limitations

We identify several limitations of our verification system. First, we assume that each data structure operation executes atomically. For this assumption to hold in concurrent settings, some form of synchronization would be required. Our current system also does not support dynamic class loading, exceptions, or inheritance. Techniques exist, however, that should make it possible to extend our modular verification approach to support such constructs [68, 11, 77]. Two limitations could be eliminated by minor extensions. We currently model numbers as algebraic quantities with unbounded precision and assume that object allocation always successfully produces a new object. While these assumptions are often used in the verification field and are typically consistent with the execution of the program, they are at variance with the full semantics of the underlying programming language. Finally, we make no attempt to verify any property related to the running time or the memory consumption of the data structure implementation. In particular, we do not attempt to verify the absence of infinite loops or memory leaks.

3.6 Summary

Jahob verifies programs written in a subset of Java and annotated with specifications written in higher-order logic. Many elements of Jahob's specification and verification condition generation approach are fairly standard. The specifications consist of specification variables, class invariants, method contracts, and annotations within the method body. In general, the abstract state of the program is described in terms of specification variables (to provide data abstraction), with method contracts expressing the behavior of the method in terms of updates to the abstract state. Class invariants express important properties preserved by operations in the class. Annotations within the method body enable the developer to express loop invariants and to refine expectations about the behavior of the method. From the annotated program, Jahob produces verification conditions using a weakest liberal precondition semantics. Together, these verification conditions ensure that methods respect their

contracts and preserve class invariants, the consistency of these invariants, and the absence of runtime errors (such as null dereference and array bounds errors).

Jahob then proves the generated verification condition formulas using a diverse collection of automated reasoning systems in an approach we call integrated reasoning. In this approach, the verification conditions are first transformed into conjunctions of smaller formulas in higher-order logic, to enable the fine-grained application of different automated provers. Then, to prove the validity of these formulas, Jahob employs an approximation technique to transform each formula into a stronger formula (to preserve soundness) in the logic subset appropriate for the specific automated reasoning system. Jahob contains interfaces to first-order provers E and SPASS, SMT provers CVC3 and Z3, decision procedures BAPA and MONA, as well as an internal syntactic prover and a simple cardinality prover. These automated reasoning systems are invoked as a cascade, with failing formulas being passed to the next prover in a user-specified order. Jahob also provides a facility to spawn provers in parallel, since the generated formulas can be proved completely independently. This facility can significantly reduce the overall proof time on multi-core machines. Where most program verification systems rely on a single monolithic prover, Jahob's integrated reasoning approach makes it possible to leverage a diversity of specialized automated reasoning systems. Using this approach, Jahob is able to verify program correctness properties that are beyond the reach of any single technique.

Chapter 4

The Integrated Proof Language

The combination of aliasing and destructive updates in imperative data structure implementations can give rise to verification conditions that are too difficult for the automated provers and decision procedures to prove without user assistance. The standard solution to this problem is for the program verification system to interface to one or more external proof assistants so that the user can interactively prove the verification conditions that fail to prove automatically [55, 22, 110, 14]. Jahob provides interfaces to the interactive theorem provers Isabelle and Coq so that users have the option of writing interactive proofs in these systems if they wish to do so.

In theory, this approach makes it possible to solve program verification problems requiring arbitrarily complex reasoning. In practice, the use of interactive theorem provers require a great deal of domain-specific expertise. Mechanically generated verification conditions can be very difficult to understand and manipulate in these external systems. The exported verification conditions may contain temporary variables generated by the system, making it difficult to determine the meaning of the formulas in their original context within the program. The formulas may also contain large numbers of irrelevant assumptions, exacerbating the difficulty of the proof task. Proving these formulas in an external system also means that the user no longer has access to the powerful automated reasoning systems integrated into Jahob, but must learn to use the (potentially) unfamiliar tools available through the interactive theorem prover.

Jahob's integrated proof language [164] addresses these issues by making it possible for developers to control proofs of program correctness properties directly within the programming and verification environment. Proof commands are embedded as annotations within the program and are verified by the underlying reasoning system as part of the original program verification workflow. Program variables within proof commands are interpreted in the standard way—as referring to the value of the variable at the given program point—eliminating confusion about the meaning of the manipulated formulas. Special constructs for identifying proved lemmas enable users to exclude unnecessary assumptions in a proof, while proof obligations generated as a result of proof commands are automatically dispatched to the automated reasoning systems, avoiding the need to write detailed proofs of supporting lemmas, or to learn the use of unfamiliar external tools.

In fact, we have found that Jahob's proof language enables developers to avoid the use of external interactive theorem provers altogether. Instead, developers simply use the Jahob proof language to resolve key choice points in the proof search space. Once these choice points have been resolved, the automated provers can then perform all of the remaining steps required to discharge the verification conditions. This approach effectively leverages the complementary strengths of the developer and the automated reasoning system by allowing the developer to communicate key proof structuring insights to the reasoning system. These insights then enable the reasoning system to successfully traverse the (in practice large and complex) proof search space to obtain formal proofs of the desired verification conditions. In practice, the developer generally needs to provide only a handful of key proof structuring insights (in the form of proof language commands) to enable the automated provers to verify a verification condition formula that they would otherwise be unable to prove.

4.1 The Proof Language Commands

Figure 3-5 presents the commands of Jahob's integrated proof language. As with Jahob specifications, these commands appear as special comments embedded within the Java source code. They are preserved by the translation to the extended guarded command language, then translated into simple guarded commands. Figure 4-1 presents the semantics of the language in terms of this translation. Note that although the proof commands can be translated into guarded commands, the translation makes use of assume commands, which would be unsound for the developer to use directly in the program. The soundness of the proof commands, on the other hand, is guaranteed by the translation. Section 4.2 and Appendix A present the proofs of soundness. Note also that some proof commands, such as localize and assuming, may enclose other proof commands, but not executable code. In general, proof commands may also enclose assert and assume.[1]

One of our goals in the design of the integrated proof language was to allow the user to provide the level of proof guidance that was comfortable for the user, and appropriate for the proof task. Specifically, we want the provers to be able to prove verification conditions with no user guidance at all if they have this capability. At the same time, we want the user to have the option of specifying every proof step explicitly if they wished to do so. Moreover, we want the language to flexibly support intermediate points at which the user and provers cooperate, with the user providing the minimum amount of guidance needed to enable the provers to complete the proof. The integrated proof language supports this wide range of behaviors by providing not only high-level commands that leverage the substantial automated reasoning power of the Jahob system, but also low-level commands that allow the user to precisely control proof steps to enable a successful proof.

[1]Enclosure of assert is always sound (see proof in Section 4.2 and Appendix A), while enclosure of assume may introduce unsoundness, and is intended only for debugging purposes.

$$
\begin{array}{rcl}
[\![\text{note } l{:}F \text{ from } \vec{h}]\!] & = & \text{assert } l{:}\,F \text{ from } \vec{h}\,; \\
 & & \text{assume } l{:}\,F \\
[\![\text{localize in } (p\,;\text{note } l{:}F)]\!] & = & (\text{skip } [\!]\,([\![p]\!]\,;\text{assert } F\,; \\
 & & \qquad \text{assume false}))\,; \\
 & & \text{assume } l{:}\,F \\
[\![\text{mp } l{:}(F \rightarrow G)]\!] & = & \text{assert } F\,;\text{assert } (F \rightarrow G)\,; \\
 & & \text{assume } l{:}\,G \\
[\![\text{assuming } l_F{:}F \text{ in } (p\,;\text{note } l_G{:}G)]\!] & = & (\text{skip } [\!]\,(\text{assume } l_F{:}\,F\,; \\
 & & \qquad [\![p]\!]\,;\text{assert } G\,; \\
 & & \qquad \text{assume false}))\,; \\
 & & \text{assume } l_G{:}\,(F \rightarrow G) \\
[\![\text{cases } \vec{F} \text{ for } l{:}G]\!] & = & \text{assert } F_1 \vee \ldots \vee F_n\,; \\
 & & \text{assert } (F_1 \rightarrow G)\,;\ldots;\text{assert } (F_n \rightarrow G)\,; \\
 & & \text{assume } l{:}\,G \\
[\![\text{showCase } i \text{ of } l{:}F_1 \vee \ldots \vee F_n]\!] & = & \text{assert } F_i\,; \\
 & & \text{assume } l{:}\,F_1 \vee \ldots \vee F_n \\
[\![\text{byContradiction } l \text{ in } Fp]\!] & = & (\text{skip } [\!]\,(\text{assume } \neg F\,; \\
 & & \qquad [\![p]\!]\,;\text{assert false}\,; \\
 & & \qquad \text{assume false}))\,; \\
 & & \text{assume } l{:}\,F \\
[\![\text{contradiction } lF]\!] & = & \text{assert } F\,;\text{assert } \neg F\,;\text{assume false} \\
[\![\text{instantiate } l \text{ with } \forall \vec{x}.F\vec{t}]\!] & = & \text{assert } \forall \vec{x}.F\,; \\
 & & \text{assume } l{:}\,F[\vec{x} := \vec{t}] \\
[\![\text{witness } \vec{t} \text{ for } l\exists \vec{x}.F]\!] & = & \text{assert } F[\vec{x} := \vec{t}]\,; \\
 & & \text{assume } l{:}\,\exists \vec{x}.F \\
[\![\text{pickWitness } \vec{x} \text{ for } l_F{:}F \text{ in } (p\,;\text{note } l_G{:}G)]\!] & = & (\text{skip } [\!]\,(\text{assert } \exists \vec{x}.F\,; \\
(\text{where } \vec{x} \text{ is not free in } G) & & \qquad \text{havoc } \vec{x}\,; \\
 & & \qquad \text{assume } l_F F\,; \\
 & & \qquad [\![p]\!]\,;\text{assert } G\,; \\
 & & \qquad \text{assume false}))\,; \\
 & & \text{assume } l_G{:}\,G \\
[\![\text{pickAny } \vec{x} \text{ in } (p\,;\text{note } l{:}G)]\!] & = & (\text{skip } [\!]\,(\text{havoc } \vec{x}\,; \\
 & & \qquad [\![p]\!]\,;\text{assert } G\,; \\
 & & \qquad \text{assume false}))\,; \\
 & & \text{assume } l{:}\,\forall \vec{x}.G \\
[\![\text{induct } l{:}F \text{ over } n \text{ in } p]\!] & = & (\text{skip } [\!]\,(\text{havoc } n\,; \\
 & & \qquad \text{assume } 0 \leq n\,; \\
 & & \qquad [\![p]\!]\,;\text{assert } F[n := 0]\,; \\
 & & \qquad \text{assert } (F \rightarrow F[n := n{+}1]) \\
 & & \qquad \text{assume false}))\,; \\
 & & \text{assume } l{:}\,\forall n.(0 \leq n \rightarrow F)
\end{array}
$$

Figure 4-1: Translating Proof Language Commands into Simple Guarded Commands

4.1.1 The Assumption Base

Conceptually, the commands of Jahob's proof language operate over a set of facts that we will refer to as the *assumption base*. A verification condition in Jahob has the form of an implication $F \rightarrow G$ where the antecedent F is a conjunction of facts. This conjunction F is the assumption base that the provers use when they attempt to prove the consequent G.

The translation of the proof language commands uses assume commands to add facts to the assumption base.[2] While it is possible for the user to introduce unsoundness by using assume commands directly in the program, the soundness of the assume commands in this context is guaranteed by the form of the translation.

Consider, for example, the translation of the assuming command. The command assuming $l_1{:}F_1$ in $(p$; note $l_2{:}F_2)$ is used to prove formulas of the form $F_1 \rightarrow F_2$. l_1 and l_2 are labels, and p proof commands. It translates into:

$$(\mathsf{skip}\ [\!]\ (\mathsf{assume}\ l_1{:}\,F_1\ ;[\![p]\!]\ ;\mathsf{assert}\ F_2\ ;\mathsf{assume}\ \mathsf{false}))\ ;\mathsf{assume}\ l_2{:}\,F_1 \rightarrow F_2$$

The translated sequence contains the following general pattern:

$$(\mathsf{skip}\ [\!]\ (c\ ;[\![p]\!]\ ;\mathsf{assert}\ F\ ;\mathsf{assume}\ \mathsf{false}))\ ;\mathsf{assume}\ G$$

The net effect of this pattern is to soundly add G to the assumption base. The pattern achieves this effect as follows. The first branch of the non-deterministic choice operator (skip) propagates the original assumption base. The second branch $(c\ ;[\![p]\!]\ ;\mathsf{assert}\ F\ ;\mathsf{assume}\ \mathsf{false})$ generates the proof obligations required to ensure that G actually holds. The assume false at the end of the second branch conceptually terminates the computation at the end of the branch so that the verification condition generator does not take the computational path through the second branch into account when generating the verification condition at the program point after the choice. This mechanism ensures that the second branch generates no proof obligations other than those required to ensure that G holds.

In effect, the second branch uses the assume false command to create a new local assumption base in which the user can guide the proof of the properties required to ensure that G holds. Because this assumption base is local, none of the assumptions or intermediate lemmas in the proof propagate through to the program point after the choice. This local assumption base mechanism therefore ensures that only G is added to the original assumption base at the program point after the translated proof language command, and that local assumptions that are only sound in the context of the proof are not propagated to the original assumption base.

In this pattern, the command c contains commands introduced as part of the translation, while p contains proof commands provided by the user and originally nested inside the proof command under translation. The command c can include commands that may modify the program state, such as assume and havoc commands. The form of the translation ensures that these commands are used in a sound way.

[2] Specifically, the verification condition generation rule in Figure 3-7 for commands of the form assume $l{:}\,F$ produces a verification condition of the form $F^{[l]} \rightarrow G$, which, in effect, adds F to the set of facts available to the provers when they attempt to prove the consequent G.

4.1.2 The Note Command

The note command is a high-level proof command used to direct Jahob to prove a specified formula. It has the general form note l:F from $\vec{h}$, which allows the user both to assign the label l to the proved lemma, as well as to specify the set of named lemmas $\vec{h}$ from which to prove the specified formula F. In general, the purpose of a note command is to identify an important intermediate lemma that the system needs to prove in order to prove a given verification condition.

The note command translates into an assert followed by an assume of the same formula. The net effect is to direct the system to prove the specified formula, then add the verified formula to the assumption base. Because Jahob proves the formula before adding it to the assumption base, the use of note is sound.

Proof Decomposition.

The note command is useful for a number of different purposes, including the ability to guide the decomposition of a proof. By directing the combined proof system to prove relevant lemmas and to add them to the assumption base using note commands, the user is effectively performing a high-level proof decomposition. The availability of these verified lemmas in the assumption base is often sufficient to guide the provers through the (usually unbounded) proof search space to successfully find a proof for the verification condition of interest.

Multiple Provers.

The note command also enables the user to decompose a proof obligation so that multiple provers (with arbitrarily narrow areas of specialization) can work together to prove it. Consider, for example, a proof obligation that involves both arithmetic reasoning and reasoning about the shape of a given data structure. By using one group of note commands to identify relevant arithmetic properties and another group of note commands to identify relevant data structure shape properties, the user can decompose the proof obligation to expose specific parts of the proof obligation to different provers. A final note command can then combine the results to deliver the complete proof obligation. A potential advantage of this approach is that the set of provers, when working together, may be able to provide sophisticated reasoning capabilities that are beyond the reach of any single general system. In general, Jahob is designed to incorporate many different provers with arbitrarily narrow areas of specialization in one complete verification system The note command is part of the infrastructure for realizing the full potential of this approach.

Controlling the Assumption Base.

Many provers perform a search over a (potentially unbounded) proof space. While it may appear that increasing the assumption base should increase the power of the prover (since the prover has more facts to work with), increasing the assumption base also increases the size of the proof search space, which may make it more difficult for

	Introduction	Elimination
Conjunction ($\wedge$)	*	*
Disjunction ($\vee$)	showCase	cases
Negation ($\neg$)	byContradiction	
Implication ($\rightarrow$)	assuming	mp
False	*	contradiction
Universal Quantification ($\forall$)	pickAny	instantiate
Existential Quantification ($\exists$)	witness	pickWitness

Figure 4-2: First-Order Logic Proof Commands. (*) denotes first-order logic proof rules that Jahob applies automatically as part of the splitting process.

the prover to find a proof. In practice, we have found that increasing the size of the assumption base may degrade the ability of the prover to find proofs for facts that it is otherwise perfectly capable of proving. The note command makes it possible for users to give names to specific facts (these facts can either be available directly in the assumption base or provable from the assumption base), then use the names in a from clause, to identify a specific set of facts that the prover should use when attempting to prove a new fact. The net effect is to eliminate irrelevant facts from the assumption base to productively focus provers on the specific facts they need to use. In general, Jahob supports the from clause in all proof commands, though for clarity, we omit it from the discussion of proof commands other than note.

4.1.3 The Localize Command

The localize command creates a new local assumption base for the proof of an arbitrary formula, then adds only this proved formula back into the original assumption base. It has the general form localize in (p ; note l:F), where F is the formula being proved, p the proof commands in the proof, and l the optional label for the proved lemma. The localize command makes it possible to use intermediate lemmas in the proof of a formula without adding these lemmas back into the original assumption base. Excluding intermediate lemmas from the original assumption base (when the lemmas are not relevant for subsequent verification conditions) can help keep the assumption base (and resulting proof search space) small enough to enable the provers to find proofs of subsequent verification conditions in an acceptable amount of time. Note that because the local assumption base is initially the same as the original assumption base, any formulas verified in the local assumption base also hold in the original assumption base. This property ensures that the command is sound.

4.1.4 First-Order Logic Commands

The integrated proof language includes a number of first-order logic proof commands which encode standard rules of natural deduction systems [126, Section 2.12], [59, Section 5.4]. When combined with the proof rules implicit in the splitting process,

$$\frac{F \quad G}{F \wedge G} \qquad \frac{F \wedge G}{F} \qquad \frac{F \wedge G}{G} \qquad \frac{\mathsf{false}}{F} \qquad \frac{}{\mathsf{true}}$$

Figure 4-3: First-Order Logic Rules Applied During Splitting

these commands give our system the completeness of first-order logic. (Note that for arbitrary higher-order logic formulas we cannot hope to obtain a proof system complete with respect to the standard models [4].) Figure 4-2 shows how the first-order logic proof commands correspond to first-order logic introduction and elimination rules. During splitting, Jahob splits top-level conjunctions of assumptions and goals, eliminating the need for conjunction introduction and elimination commands. Jahob also incorporates the standard rule for deriving any formula when false is one of the assumptions. Figure 4-3 shows the rules that Jahob automatically applies as part of the splitting process.

The Assuming Command

The assuming command encodes the implication introduction rule of first-order logic. It allows the user to prove a fact of the form $F \rightarrow G$ by hypothesizing F, then deriving G. It has the general form assuming F in $(p$; note $G)$, where $F \rightarrow G$ is the formula being proved, while p consists of the proof statements that derive G given that F holds. The command effectively creates a new local assumption base, which contains all the facts in the original assumption base, plus F. The sequence p of proof commands can then use this local assumption base to derive consequences of F and the previously known facts to guide the proof of G. At the end of the proof, only the proved lemma $F \rightarrow G$ is added to the original assumption base, ensuring soundness.

The assuming command is necessary for decomposing the components of an implication. Without such a command, many intermediate lemmas in a proof of a formula $F \rightarrow G$ would themselves be implications that could not be decomposed, increasing the difficulty of the proof task. In practice, we have found that the assuming command is particularly useful when G is complex, as the provers often fail to find the proof in such cases without guidance.

The Mp Command

The mp command is the dual of assuming, and encodes the *modus ponens* rule of inference. It allows the user to conclude a goal G by directing Jahob to prove F and $F \rightarrow G$, then soundly adding G to the assumption base.

It is also possible to simulate the effects of the mp command using a sequence of note commands, as follows:

$$\mathsf{note}\ l_1{:}F; \quad \mathsf{note}\ l_2{:}(F \rightarrow G); \quad \mathsf{note}\ G\ \mathsf{from}\ l_1, l_2$$

We find that this use of the note command works well in practice for the automated provers used by Jahob. However, the mp command has certain advantages over note.

It is more concise, provides precise control over the proof step, and does not introduce additional facts into the assumption base. Most importantly, it is not dependent on the ability of the automated provers to recognize the need to apply the *modus ponens* rule of inference to obtain the proof.

The ShowCase Command

The showCase command encodes the disjunction introduction rule. It allows the user to soundly establish a fact of the form $F_1 \vee \ldots \vee F_n$ by proving a case F_i in the disjunction. The command instructs Jahob to prove the ith case of the disjunction, then soundly adds $F_1 \vee \ldots \vee F_n$ to the assumption base.

As with the mp command, it is possible to simulate the effects of the showCase command using note, modulo the ability of the provers to handle disjunction:

$$\mathsf{note}\ l{:}F_i;\quad \mathsf{note}\ (F_1 \vee \ldots \vee F_n)\ \mathsf{from}\ l$$

The use of the showCase command over note offers similar advantages as in the case of mp, with the primary advantage being the ability to perform the desired proof step regardless of any limitations in the automated provers' abilities to perform disjunction introduction.

The Cases Command

The cases command allows the user to prove an arbitrary formula G using case analysis. It enables the user to specify a set $F_1, \ldots, F_n$ of cases to consider in the proof of G, ensures that the set of cases is complete and that the goal holds in each case (i.e. instructs Jahob to prove $F_1 \vee \ldots \vee F_n$ and $F_i \rightarrow G$ for $1 \leq i \leq n$), then adds the proved goal G to the assumption base.

In principle, it is also possible to perform case analysis using a sequence of note commands. For example:

$$\mathsf{note}\ l_1{:}(F \rightarrow G);\quad \mathsf{note}\ l_2{:}((\neg F) \rightarrow G);\quad \mathsf{note}\ G\ \mathsf{from}\ l_1, l_2$$

In practice, however, we find that the provers are sometimes unable to derive the final goal G from the set of cases, possibly due to an inability to recognize the need for case analysis in the given situation. The cases command frees the user from dependence on the particular abilities of the provers to find a proof by directly applying case analysis in a sound way.

The PickAny Command

The pickAny command encodes the universal introduction rule. It allows the user to prove a fact of the form $\forall x.G$ by considering an arbitrary x, then proving that G holds for x. The general form of the command is pickAny x in $(p\,;\mathsf{note}\ G)$, where p is a sequence of proof commands for guiding the proof of G. The command causes the universally quantified variable x to become visible inside the sequence p as a local

specification variable with an arbitrary value. The user can therefore state lemmas (within p) that involve x as a fixed variable. Note that x is also not a quantified variable in the verification conditions that result, further simplifying the theorem proving task.

The effect of the pickAny command is to create a new local assumption base that starts out as a copy of the original assumption base, but with a local specification variable x whose value is undefined. Intermediate lemmas added to the local assumption base in the course of the proof remain within the local scope and are not propagated to the original. At the conclusion of the proof, $\forall x.G$ is soundly added to the original assumption base, which is otherwise unchanged.

The pickAny command can also be used with an optional suchThat clause, as in pickAny x suchThat F in $(p$; note $G)$. This combination serves as a shorthand for a pickAny block enclosing an assuming block, and is used for proving formulas of the form $\forall x.F \rightarrow G$, such as class invariants, replacing the more verbose proof command sequence pickAny x in (assuming F in $(c$; note $G)$; note $G)$.

The Instantiate Command

The instantiate command encodes the universal elimination rule. It allows the user to establish a fact of the form $G[\vec{x} := \vec{t}]$ by proving a fact of the form $\forall \vec{x}.G$. It instructs Jahob to prove $\forall \vec{x}.G$, then soundly adds $G[\vec{x} := \vec{t}]$ to the assumption base.

It is possible to simulate the effects of the instantiate command using note, as the automated provers used in Jahob are adept at instantiating a universally quantified formula when given the appropriate term:

$$\text{note } l{:}\forall \vec{x}.G; \quad \text{note } G[\vec{x} := \vec{t}] \text{ from } l$$

However, the more common use of the note command in lieu of instantiate occurs when combining multiple proof steps, which is possible if the automated provers are capable of performing each proof step automatically when given the goal and the necessary facts. Specifically, either of the following two sequences may be used to prove a goal of the form $G[\vec{x} := \vec{t}]$ by instantiating a universally quantified formula and applying *modus ponens* to the result:

$$\text{note } l_1{:}\forall \vec{x}.F \rightarrow G; \quad \text{note } l_2{:}F[\vec{x} := \vec{t}]; \quad \text{note } G[\vec{x} := \vec{t}] \text{ from } l_1, l_2$$

$$\text{instantiate } \forall \vec{x}.F \rightarrow G \text{ with } \vec{t}; \quad \text{mp } F[\vec{x} := \vec{t}] \rightarrow G[\vec{x} := \vec{t}]$$

Both sequences work well in practice. The sequence that uses note takes advantage of the ability of the automated provers to perform both the instantiation and implication elimination in a single step using the third note statement, which may result in a more concise proof when facts l_1 and l_2 are either already named or necessary for other proof commands. Where this is not the case, the use of the more specific instantiate and mp commands is more concise. In general, the more specific commands allow more precise control over the proof steps, and produce a proof that has greater independence from the capabilities and limitations of the specific automated provers in use.

The Witness Command

The witness command encodes the existential introduction rule. It allows the user to establish a fact of the form $\exists \vec{x}.G$ by proving a fact of the form $G[\vec{x} := \vec{t}]$. It instructs Jahob to prove $G[\vec{x} := \vec{t}]$, then soundly add $\exists \vec{x}.G$ to the assumption base.

In principle, it should be possible to provide a witness for an existentially quantified formula using the following sequence of note commands:

$$\text{note } l{:}G[\vec{x} := \vec{t}]; \quad \text{note } \exists \vec{x}.G \text{ from } l$$

In practice, we find that the automated provers are often unable to derive the existentially quantified goal even when the witness is provided, as in the sequence of note commands above. Thus the witness command is critical to the ability of the system to prove existentially quantified goals.

The PickWitness Command

The pickWitness command encodes the existential elimination rule. It allows the user to instantiate a formula of the form $\exists \vec{x}.F$ (i.e., eliminate the existential quantifier and name the values that satisfy the constraint F) in a new local assumption base, guide the proof of a formula G, then add the proved goal G back into the original assumption base. To ensure soundness, $\vec{x}$—the variable(s) with which the user is instantiating $\exists \vec{x}.F$—must not be free in G. The general form of the pickWitness command is pickWitness $\vec{x}$ for F in $(p\,;\text{note}\,G)$.

By enabling the user to name values of $\vec{x}$ for which the constraint F is true, the pickWitness command makes it possible to replace an existentially-quantified formula with an instantiated version, then state additional facts about the named values. This functionality broadens the applicability of provers with limited ability to reason about existentially quantified formulas. Without the pickWitness command, every subgoal G_i that depended on the constrained values would have to have the form $\exists \vec{x}.(F \rightarrow G_i)$. Such a subgoal is beyond the reach of any prover that cannot reason effectively about existentially quantified formulas. The pickWitness command enables the user to soundly eliminate the existential quantifier, thereby transforming existentially quantified proof goals into a form that the provers can handle more effectively.

The ByContradiction Command

The byContradiction command allows the user to prove an arbitrary formula F using proof by contradiction. It enables the user to add $\neg F$ to a new local assumption base, then use this assumption base to guide the proof of false. The verified formula F can then be soundly added to the original assumption base. The user can also use this command to perform negation introduction by directing Jahob to prove a formula of the form $\neg F$.

The Contradiction Command

The contradiction command allows the user to derive false from a contradiction. It makes it possible for the user to guide the proof of a formula F and its negation $\neg F$ to soundly conclude false. (Specifically, it instructs Jahob to prove F and $\neg F$, then adds false to the assumption base.) Although we did not use the contradiction (or the byContradiction) command in the data structures that we verified for this thesis, we provide these commands in the proof language for completeness.

4.1.5 The Induct Command

The induct command allows the user to prove a fact of the form $\forall n.(0 \leq n \rightarrow F)$ using mathematical induction. Specifically, it allows the user to select an arbitrary value of n for which $0 \leq n$ holds, guide the proof of the base case $F[n := 0]$ and the inductive case $F \rightarrow F[n := n+1]$, then add the proved goal into the assumption base. The introduction of the constraint $0 \leq n$ makes it possible to simulate mathematical induction over natural numbers using integers.

The general form of the command is induct n over F in p, where p is a sequence of proof commands for guiding the proof of the base and inductive cases. The induct command creates a new local assumption base that is a copy of the original assumption, but with a local specification variable n which has an arbitrary value greater than zero. The proof commands p operate over this local assumption base, where n is a fixed variable. The induct command then directs the system to prove both the base and inductive cases, then add only the proved goal $\forall n.(0 \leq n \rightarrow F)$ back into the original assumption base.

The induct command is particularly important because the automated provers are unable to perform proofs that require mathematical induction, even though they may be able to derive many of the supporting facts in the proof. The induct command provides the inductive schema necessary to soundly derive a fact using induction, thereby reducing the inductive proof to proof steps that the automated provers are equipped to handle. Without this command, the only recourse for the user would be to interactively prove the desired goal using an external proof assistant.

4.2 Soundness

Figure 4-1 gives the translation rules that define the semantics of our proof language commands. We show the soundness of each of these rules using properties of weakest liberal preconditions [6]. The proof demonstrates that the verification conditions generated (using weakest liberal preconditions) with the proof commands present in the program are stronger than the verification conditions that would be generated for the program without proof commands. As a result, by proving the verification condition formulas for the program containing proof commands, we are proving a stronger condition that guarantees the correctness of the original program.

The main idea is as follows. Without proof commands in the annotated program, the system produces a verification condition formula G_1, that guarantees the

correctness of the program. With proof commands, the system produces a different verification condition formula G_2. To ensure soundness, we need to show that the verification condition proved by the system, G_2, is stronger than the original verification condition, G_1. In other words, we need to show that $G_2 \rightarrow G_1$. We prove this by induction on the structure of the proof commands.

We define the relation $\sqsubseteq$ such that $c \sqsubseteq c'$ if and only if $\mathsf{wlp}(c, F) \rightarrow \mathsf{wlp}(c', F)$ for all formulas F. In this case we say that c is stronger than c'. Let skip be the no-op command. We show that $p \sqsubseteq \mathsf{skip}$ for all p by induction on p. This is sufficient for soundness because it ensures that any property provable for the annotated program containing proof language commands also holds in the unannotated program (which is equivalent to the annotated program with all proof commands replaced with skip).

The induction hypothesis is $\mathsf{wlp}([\![p]\!], H) \rightarrow H$, where H is an arbitrary formula. For each proof language command p, we apply the translation rules in Figure 4-1, the rules of weakest liberal preconditions in Figure 3-7, the induction hypothesis, and the standard rules of logic to show that $\mathsf{wlp}([\![p]\!], H) \rightarrow H$—i.e., that p is stronger than skip.

As a sample inductive step, consider the assuming command. By applying the translation rule for assuming, the rules of weakest liberal preconditions, and the standard rules of logic, we obtain:

$$\begin{aligned} & \mathsf{wlp}([\![\mathsf{assuming}\ F\ \mathsf{in}\ (p\,;\mathsf{note}\ G)]\!], H) \\ = \ & \mathsf{wlp}((\mathsf{skip}\ [\,](\mathsf{assume}\ F\,;[\![p]\!]\,;\mathsf{assert}\ G\,;\mathsf{assume\ false}))\,; \\ & \qquad \mathsf{assume}\ (F \rightarrow G)), H) \\ = \ & ((F \rightarrow G) \rightarrow H) \wedge (F \rightarrow \mathsf{wlp}([\![p]\!], G)) \end{aligned}$$

According to the induction hypothesis, $((F \rightarrow G) \rightarrow H) \wedge (F \rightarrow \mathsf{wlp}([\![p]\!], G))$ implies the formula $((F \rightarrow G) \rightarrow H) \wedge (F \rightarrow G)$, which in turn implies H. Consequently, assuming is stronger than skip and its translation is sound.

The proofs for the other commands are similar. See Appendix A for the complete soundness proofs for all the proof language commands and for assert.

4.3 Proof Reuse and Parameterization

Jahob uses standard Java methods as a mechanism for reusing and parameterizing proofs. A *proof method* is simply a Java method that contains only proof commands. The requires clause of the proof method specifies the necessary precondition for the proof, while the ensures clause specifies the postcondition that the proof guarantees. By definition, a proof method has no modifies clause, since proof commands do not modify the program state.

By enclosing a proof in a proof method, the user can parameterize the proof using Java method parameters, and apply it anywhere the proof method may be invoked using a method call. Because Jahob handles all method calls (including those that invoke proof methods) using standard assume-guarantee reasoning, it checks that method preconditions are met at invocation sites, ensuring that proofs are only applied when the precondition of the proof holds. The postcondition of the proof

can then be soundly assumed to hold on return from the method, with soundness guaranteed by the verified method. Jahob verifies methods (including proof methods) by checking that they conform to their specification. For proof methods, this ensures the correctness of the proof.

Although Jahob does not allow proof commands to enclose method calls, it would not violate soundness if Jahob were to relax this restriction for proof methods. The main concern is that it is not sound for proof commands to enclose executable Java code, which proof methods do not contain. To enable this feature in a safe way, it would be necessary for Jahob to perform an additional check to ensure that a method call inside a proof command does indeed invoke a proof method.[3] It would also be necessary to extend the soundness proof in Appendix A to take method calls into account.

4.3.1 Verification Feedback

When the combined reasoning system is unable to prove a verification condition formula, Jahob provides the developer with the failed formula. As part of the verification condition generation process, Jahob embeds comments within the formulas that identify the relevant properties and the path through the program that the system is verifying. Because of the nature of the weakest liberal precondition algorithm that Jahob uses to generate verification conditions, failed formulas always encode the path through the program that failed to verify. As a result, the returned formula effectively provides the developer with the property that failed to prove (e.g. a class invariant, a postcondition, or loop invariant), as well as a counterexample for the failure. This counterexample can then be used to determine the proof commands needed to produce a successful proof.

4.4 Summary

Jahob's integrated proof language is a declarative proof language that enables users to guide the combined reasoning system in proving properties that are otherwise beyond the reach of the automated provers. It extends the assertion mechanism available in most program verification systems by allowing the user to embed proof commands as special comments directly in the annotated program. (This mechanism also makes it possible to parameterize and reuse existing proofs by encapsulating proofs within Java methods.) Proof commands produce verification conditions that are automatically dispatched to the automated reasoning systems, allowing users to provide only the minimum amount of guidance needed to lead the provers to a successful proof. The remaining proof steps are handled automatically by the combined reasoning system. The proof language also supports commands that enable the precise specification of proof steps, making it possible to write detailed proofs within the language, and flexibly supporting intermediate points in the design space

[3]This can be done in two ways. Specifying an annotation to denote proof methods would enable modular checking, while a fixed-point interprocedural analysis would avoid the need for annotations.

between completely manual and fully automated proofs, depending on the abilities of the automated reasoning systems. In this chapter, we presented the proof commands in our language, defined the semantics of the language by translation into simple guarded commands, and proved the soundness of our translation according to weakest liberal precondition semantics. We have implemented our proof language in the Jahob program verification system, and used it to verify the data structures in this thesis.

Chapter 5

Priority Queue

The next several chapters describe three of the data structures we have verified using Jahob. Chapter 6 describes an association list, while Chapter 7 describes a hash table data structure. In this chapter, we describe a priority queue. The priority queue data structure is an example of an array-based data structure that also maintains important ordering properties. It is one of the most difficult examples in our collection of verified data structures. We use it to illustrate some of the techniques for specifying and verifying array-based data structures with complex correctness properties.

Our PriorityQueue class implements a priority queue using a binary heap stored in an array [40]. Instead of using explicit pointers, the tree structure of the priority queue is defined implicitly by the following mathematical relationships between the array indices of the parent and child nodes.

$$\begin{aligned} child_{\mathsf{left}}(i) &= 2i+1 \\ child_{\mathsf{right}}(i) &= 2i+2 \\ parent(i) &= floor\left(\frac{i-1}{2}\right) \end{aligned}$$

The implementation maintains the binary heap property; every node in the tree has a priority that is greater than or equal to that of its children. As a result, the root node of the tree is guaranteed to have the highest priority.

5.1 The Concrete State

Figure 5-1 presents part of the PriorityQueue class as well as the Node class that stores the objects in the queue.[1] The concrete state of a PriorityQueue object consists of an array queue, which stores the contents of the queue using Node objects, and an integer length, which stores the number of objects currently in the queue. The

[1]Formulas in Jahob specifications and proof language statements contain mathematical notation for concepts such as set union ($\cup$) and universal quantification ($\forall$). Developers can enter these symbols in Jahob input files using X-Symbol ASCII notation, and view them in either ASCII or mathematical notation using the ProofGeneral editor mode for emacs [5].

```
public /*: claimedby PriorityQueue */ class Node {
    public Object ob;
    public int pr;
}

public class PriorityQueue
{
    private Node[] queue;
    private int length;
    /*:
      public specvar init :: bool = "False";
      public specvar contents :: "(obj * int) set";
      public specvar capacity :: int;
      specvar nodes :: "obj set";

      vardefs "init == (queue ≠ null)";
      vardefs "nodes == {n.(∃i. 0 ≤ i ∧ i < length ∧ n = queue.[i])}";
      vardefs "contents ==
               {(x,y).(∃n.n ∈ nodes ∧ n..ob = x ∧ n..pr = y)}";
      vardefs "capacity == queue..Array.length";

      invariant OrderedInv: "init →
      (∀i j.(0 ≤ i ∧ i < length ∧ 0 ≤ j ∧ j < length ∧
      (j=2*i+1 ∨ j=2*i+2)) → queue.[i]..pr ≥ queue.[j]..pr)";
      invariant CardInv: "init → length = card(contents)";
      invariant CapacityInv: "init → 0 < capacity";
      invariant InitialLength: "¬init → length = 0";
      invariant LengthInv:
      "init → 0 ≤ length ∧ length ≤ queue..Array.length";
      invariant NonNullInv:
      "init → (∀i. 0 ≤ i ∧ i < length → queue.[i] ≠ null)";
      invariant DistinctInv: "init →
      (∀i j.(0 ≤ i ∧ i < length ∧ 0 ≤ j ∧ j < length ∧
      queue.[i] = queue.[j]) → i = j)"
      invariant NullInv: "init →
      (∀i. length ≤ i ∧ i < queue..Array.length → queue.[i] = null)";
      invariant HiddenInv: "init → queue ∈ hidden";
      invariant InjInv:
      "∀x y. x..queue = y..queue ∧ x..queue ≠ null → x = y";
      invariant ContentsInj:
      "∀e1 e2. e1 ∈ nodes ∧ e2 ∈ nodes ∧ e1 ≠ e2 →
       (e1..ob ≠ e2..ob ∨ e1..pr ≠ e2..pr)"; */
    ...
}
```

Figure 5-1: PriorityQueue Example

concrete state of a Node object consists of an object field ob, which stores an object in the queue, and an integer field pr, which stores that object's corresponding priority. The claimedby annotation in the declaration of the Node class indicates that only methods in the PriorityQueue class may modify the fields of Node objects. We retain the public access designation for the fields of the class so that the Java compiler would allow the PriorityQueue class to access them, but augmenting Jahob with support for static nested classes would make this unnecessary.

5.2 The Abstract State

The abstract state of the priority queue is specified using dependent specification variables, declared using the specvar keyword in Figure 5-1. The declarations indicate that the public abstract state of a PriorityQueue object consists of contents, which represents the contents of the queue as a set of object-priority pairs; capacity, the maximum number of objects that the queue is currently able to hold; and init, a boolean flag which indicates whether the queue is initialized. The initial value of init is set to false to ensure that class invariants are valid in the initial program state. In general, an init flag is useful for specifying data structures whose invariants are not guaranteed to hold until the constructor has executed. In such a case, class invariants would have the form init $\rightarrow F$, where F is an invariant of an initialized object. Instance methods would have init as a conjunct in the precondition, while constructors would have init as a conjunct in the postcondition.

A PriorityQueue object also has private abstract state, which consists of nodes, the set of Node objects in use by the queue. Private abstract state is useful for reasoning about the implementation of a data structure at a higher level of abstraction without exposing implementation-specific details to the data structure clients. In this case, the specification variable nodes serves as a convenient and intuitive shorthand for its definition, making it possible to write Jahob formulas within the PriorityQueue class that are more concise and easier to understand.

5.3 The Abstraction Function and Invariants

For dependent specification variables, the vardefs declarations comprise the abstraction function that defines the relationship between the abstract and concrete states. In the PriorityQueue class, the vardefs for the specification variable init indicates that init is true if and only if the array queue is non-null. The vardefs for nodes defines nodes as the set of array elements consisting of all queue.[i] for which $0 \leq$ i $<$ length.[2] The vardefs for contents defines contents as a set of pairs, where each pair (x,y) corresponds to a node n in nodes, where x is n..ob and y is n..pr.[3] Finally, the vardefs for capacity defines the capacity of the priority queue as the length of the array queue.

[2]The expression queue.[i] is Jahob's notation for the Java expression queue[i].

[3]The Jahob formula expressions n..ob and n..pr correspond to the Java expressions n.ob and n.pr, respectively.

Although dependent specification variables are conceptually simply shorthands for their respective definitions, they also serve as an abstraction mechanism. Contracts for public methods can be written in terms of specification variables without exposing the data representation, which would not be the case if their definitions were used directly in the method contract.

The invariant declarations in the PriorityQueue class supplement the abstraction function provided by the vardefs declarations with data structure invariants that express additional constraints on the valid abstract and concrete states of the priority queue. The PriorityQueue class contains the following invariants.

- The OrderedInv invariant expresses the heap property. It indicates that the priority of every node in an initialized priority queue is greater than or equal to the priority of its children.

- The CardInv invariant gives the relationship between the length field of an initialized PriorityQueue object and the abstract state of the queue. It indicates that length corresponds to the cardinality of contents.

- The CapacityInv invariant indicates that the capacity of an initialized queue is greater than zero.

- The InitialLength invariant indicates that the length of an uninitialized queue is zero.

- The LengthInv invariant indicates that the length of an initialized priority queue is in the range $[0, \text{queue..Array.length})$. This invariant is used to guarantee that accesses to queue are always within bounds. It refers to the fully-qualified name Array.length to distinguish the Array.length field from PriorityQueue.length.

- The NonNullInv invariant indicates that, for an initialized priority queue, all elements of queue that are in use are non-null. This invariant is used to guarantee the lack of null dereferences when accessing elements of queue.

- The DistinctInv invariant indicates that, for an initialized priority queue, all elements of queue that are in use are distinct.

- The HiddenInv invariant indicates that, for an initialized priority queue, queue is in the set Priority.hidden. This invariant makes it possible for public methods that modify arrays referred to by queue to soundly omit arrayState from their respective modifies clauses. Jahob ensures that hidden objects remain private to the class, and generates frame conditions that permit the modification of hidden arrays.

- The InjInv invariant indicates that queue is injective (i.e. no two priority queue objects share the same queue array). This property is necessary to ensure that when the queue of a PriorityQueue object is modified, all other PriorityQueue objects remain unchanged.

- The ContentsInj invariant indicates that no two nodes in the queue may store the same object and priority. This property is related to the fact that the priority queue exports a map interface that maps objects to priorities. To implement a queue that could contain more than one occurrence of the same object-priority pair, either its interface would be a set of triples, where each triple denotes an object, a priority, and the number the times that object-priority pair occurs in the queue, or else any remove operation would have to remove all instances of the same object-priority pair from the queue. Otherwise, it would not be possible to completely characterize the behavior of the remove operation.

Although the vardefs declarations and data structure invariants of the PriorityQueue class do not contain explicit references to the receiver object this, Jahob automatically resolves implicit references, as is standard in Java. For example, Jahob automatically resolves init in CardInv to this..init. When an invariant contains either an explicit or implicit reference to the receiver object, Jahob automatically quantifies that invariant over all allocated instances of the class. For example, CardInv contains init, length, and contents, all of which implicitly refer to this. Since we want to ensure that CardInv holds for all allocated instances of the PriorityQueue class, Jahob automatically quantifies CardInv to produce the following formula, where alloc refers to the set of all allocated objects.

$$\forall \mathsf{this.this} \in \mathsf{PriorityQueue} \wedge \mathsf{this} \in \mathsf{alloc} \wedge \mathsf{this..init} \rightarrow \mathsf{this..length} = \mathsf{card}(\mathsf{this..contents})$$

Jahob performs the same transformation for all invariants that contain implicit references to the receiver.

Note that many of the invariants in the PriorityQueue class have counterparts in our other data structure implementations. In general, data structures whose contents are ordered will have ordering invariants like OrderedInv. Data structures that support size operations will have an invariant like CardInv that expresses the relationship between the number of elements stored in the data structure and the field that keeps track of this value. Data structures implemented using arrays will have invariants like LengthInv, for ensuring that array accesses are within bounds, and some form of HiddenInv, to enable more accurate method contracts. Regardless of whether the data structure is array-based, most will require some sort of injectivity invariant, expressing the lack of sharing between different instances (such as InjInv), and between different components of the same instance (such as DistinctInv). In some cases, the same invariant may express both properties. Most data structures will also require some form of NonNullInv to ensure the lack of null dereferences. Although many of these invariants seem obvious, it often took a failing verification condition before we realized that they were part of our assumptions about the implementation. Because there is so much overlap between the types of invariants in different data structures, it may be possible to leverage common patterns to automatically generate these invariants.

```
public PriorityQueue(int initialCapacity)
/*: requires "¬init ∧ 0 < initialCapacity"
    modifies init, contents, capacity
    ensures "init ∧ contents = ∅ ∧ capacity = initialCapacity" */
{ ... }

public void add(Object o1, int p1)
/*: requires "init ∧ o1 ≠ null"
    modifies contents, capacity
    ensures "contents = old contents ∪ {(o1,p1)} ∧
             capacity ≥ old capacity" */
{ ... }

public void clear()
/*: requires "init"
    modifies contents
    ensures "contents = ∅" */
{ ... }

public Object peek()
/*: requires "init"
    ensures "(contents = ∅ → result = null) ∧
             (contents ≠ ∅ →
               (∃p. (result,p) ∈ contents ∧
                 (∀x y.(x,y) ∈ contents → p ≥ y)))" */
{ ... }

public Object poll()
/*: requires "init"
    modifies contents
    ensures "(old contents = ∅ →
               contents = old contents ∧ result=null) ∧
             (old contents ≠ ∅ →
                (∃p. (result,p) ∈ old contents ∧
                  contents = old contents − {(result,p)} ∧
                  (∀x y. (x,y) ∈ old contents → p ≥ y)))" */
{ ... }

public int size()
/*: requires "init"
    ensures "result = card(contents)" */
{ ... }
```

Figure 5-2: Exported Operations of the PriorityQueue Class

5.4 Method Contracts for Public Methods

Method contracts capture the behavior of methods in terms of their effect on the abstract and concrete state of the program. Figure 5-2 presents the method contracts for all the public methods of the PriorityQueue class. These methods constitute the interface that the PriorityQueue class exports to its clients. The interfaces for these methods are modeled after the interfaces for the corresponding methods in java.util.PriorityQueue.

5.4.1 The PriorityQueue Constructor

The PriorityQueue constructor is a public constructor that takes an initialCapacity and creates a new, empty, initialized PriorityQueue object. The requires clause indicates the constructor must be invoked on an uninitialized priority queue and that the initialCapacity parameter must be greater than zero. The modifies clause indicates that the constructor may modify the init, contents, and capacity components of the priority queue under construction. The ensures clause indicates that the resulting priority queue is initialized and empty, with capacity equal to initialCapacity. By requiring that the priority queue under construction be uninitialized, the constructor may omit the initialization of the PriorityQueue.length field, which is known to be zero under Java semantics, and ensured by the InitialLength invariant, which applies to uninitialized priority queues.

Note that the init specification variable is user-defined, and, as such, must be explicitly mentioned in the precondition and postcondition of the constructor, as Jahob has no special knowledge about its meaning. However, this pattern is quite common, and it may be useful to make init a system-defined specification variable that is implicitly conjoined to the preconditions and postconditions of constructors and methods, either negatively or positively, as appropriate.

5.4.2 The add() Method

The add() method is a public instance method that adds an object o1 to the priority queue with priority p1, and may also increase the capacity of the queue. The requires clause indicates that the method must be invoked on an initialized priority queue, and that the argument o1 must be non-null. The modifies clause indicates that the method may change both the contents and capacity components of the abstract state of the queue. The ensures clause indicates that the method adds the pair (o1,p1) to the contents of the queue and that the capacity of the queue after the method executes is greater than or equal to the capacity before the method executes. The expressions old contents (and old capacity) in the ensures clause refer to the value of contents (and capacity) before the method is invoked.

Note that we modeled the priority queue with a notion of the current maximum capacity. It is also possible to strengthen or weaken the method contract, either by specifying the exact conditions under which the capacity changes, and/or how it changes, or to remove the capacity component of the abstract state altogether from the

specifications. The weaker specification provides less information, but the stronger specification may expose details of the implementation that the developer may wish to change down the line. In this example, we verified an interface that provides some additional information but does not excessively constrain the implementation.

5.4.3 The clear() Method

The clear() method is a public instance method that removes all the entries from the priority queue. The requires clause indicates that clear() must be invoked on an initialized priority queue. The modifies clause indicates that clear() may modify the abstract state component contents of the given queue. The ensures clause indicates that invoking clear() results in an empty priority queue.

5.4.4 The peek() Method

The peek() method is a public instance method that returns an element in the queue with the maximal priority. The requires clause indicates that peek() must be invoked on an initialized priority queue. The ensures clause indicates that if the priority queue is empty, then peek() returns null. If the priority queue is not empty, then peek() returns an element in the queue that corresponds to the highest priority. There is no modifies clause because peek() does not modify the program state. The keyword result in the ensures clause refers to the method's return value. Our priority queue implementation allows for multiple objects in the queue with the same priority. So while peek() will return an element with the maximal priority, there may be more than one such element.

5.4.5 The poll() Method

The poll() method is a public instance method that, like the peek() method, returns an element in the queue with the maximal priority. But, unlike the peek() method, it also removes the returned element from the queue. The requires clause indicates that poll() must be invoked on an initialized priority queue. The modifies clause indicates that poll() may change the contents of the queue. The ensures clauses identifies two possible cases. In the first case, the contents of the priority queue is empty. In this case, contents remains unchanged, and the method returns null. In the second case, the priority queue is not empty. In this case, the method removes and returns an object from the queue with the maximal priority.

5.4.6 The size() Method

The size() method is a public instance method that returns the number of objects currently in the queue. The requires clause indicates that the method must be invoked on an initialized priority queue. The ensures clause indicates that the method returns the cardinality of the queue's contents. The size() method has no modifies clause because it does not change the state of the queue.

5.5 Method Contracts for Private Methods

Figures 5-3 and 5-4 present the method contracts for the private methods of the PriorityQueue class. These methods implement various functionalities that are needed to support the public operations of the priority queue data structure.

5.5.1 The parent() Method

The parent() method is a private static method that computes the parent index for a node. It takes an index i and returns the index corresponding to the parent of the node at i. It is used by the private method addOnly() to access the parent of a newly added node, to determine whether to move the added node higher up in the binary heap. The requires clause of the method contract for parent() indicates that it must be invoked on an index that is greater than zero. This is because zero corresponds to the root node, which has no parent, while negative indices are invalid.

The ensures clause indicates that the index returned by parent() is in the range $[0, i)$ and is the index of the parent of the node at i. Although the former property is a consequence of the latter, it is needed by all the methods that invoke parent() to ensure that the result is a valid index into the queue array. Including this property in the postcondition of parent() results in Jahob verifying it once for parent(), instead of verifying it once for every method that calls parent(). The ensures clause also indicates that parent() does not allocate any objects. This property is necessary in the verification of methods that call parent(), to prove properties that depend on the value of alloc. These properties include frame conditions as well as many of the priority queue invariants, which are quantified over objects in alloc. The parent() method has no modifies clause as it does not change the program state.

5.5.2 The left() Method

The left() method is a private static method that takes an index i and returns the index corresponding to the left child of the node at i. The private method heapify() invokes left() to access the left child of a given node (to determine whether that node needs to be swapped with its left child to restore the binary heap property after the removal of a node). The requires clause indicates that left() must be invoked on an index that is greater than or equal to zero. The ensures clause indicates that left() returns an index that is greater than i and corresponding to the left child of the node at i. It also indicates that left() does not allocate any objects. There is no modifies clause because the left() method does not change the program state.

5.5.3 The right() Method

The right() method is the dual of left(). It is a private static method that takes an index i and returns the index corresponding to the right child of the node at i. The private method heapify() invokes right() to access the right child of a given node (to determine whether that node needs to be swapped with its right child to restore the

```
private static int parent(int i)
/*: requires "0 < i"
    ensures "0 ≤ result ∧ result < i ∧
             (i = 2 * result + 1 ∨ i = 2 * result + 2) ∧
             alloc = old alloc" */
{ ... }

private static int left(int i)
/*: requires "0 ≤ i"
    ensures "result = 2 * i + 1 ∧ alloc = old alloc" */
{ ... }

private static int right(int i)
/*: requires "0 ≤ i"
    ensures "result = 2 * i + 2 ∧ alloc = old alloc" */
{ ... }

private boolean contains(Object o1, int p1)
/*: requires "init ∧ o1 ≠ null ∧ theinvs"
    ensures "result = ((o1, p1) ∈ contents) ∧ theinvs" */
{ ... }

private void resize()
/*: requires "init ∧ theinvs"
    modifies queue, capacity, arrayState
    ensures "length < capacity ∧
             (∀x. x ∈ PriorityQueue → x..contents = old (x..contents)) ∧
             (∀x. x ∈ PriorityQueue → x..init = old (x..init)) ∧
             (∀a i. a ≠ queue → a.[i] = old (a.[i])) ∧ theinvs" */
{ ... }

private void addOnly(Object o1, int p1)
/*: requires "init ∧ o1 ≠ null ∧ (o1, p1) ∉ contents ∧
              length < capacity ∧ theinvs"
    modifies contents, length, arrayState, ob, pr
    ensures "contents = old contents ∪ {(o1, p1)} ∧
             length = old length + 1 ∧
             (∀x. x ∈ old alloc ∧ x ∈ PriorityQueue ∧ x ∉ hidden ∧
              x ≠ this → x..contents = old (x..contents)) ∧
             (∀a i. a ≠ queue → a.[i] = old (a.[i])) ∧ theinvs" */
{ ... }
```

Figure 5-3: Private Methods of the PriorityQueue Class (continued in Figure 5-4)

```
private void heapify(int i)
/*: requires "init ∧ 0 ≤ i ∧ i < length ∧
              comment ''GlobalOrderingPre''
              (∀k j.(0 ≤ k ∧ k < length ∧ k ≠ i ∧ 0 < j ∧ j < length ∧
                ((j = 2*k + 1) ∨ (j = 2*k + 2)) →
                  queue.[k]..pr ≥ queue.[j]..pr)) ∧
              comment ''LocalOrderingPre''
              (∀x.((0 ≤ x ∧ (i = 2*x + 1 ∨ i = 2*x + 2)) →
                (((2*i + 1 < length) →
                  queue.[x]..pr ≥ queue.[(2*i + 1)]..pr) ∧
                 ((2*i + 2 < length) →
                  queue.[x]..pr ≥ queue.[(2*i + 2)]..pr)))) ∧
              comment ''OrderedFrame''
              (∀pq. pq ∈ PriorityQueue ∧ pq ∈ alloc ∧ pq..init ∧
                pq ≠ this →
                (∀i j. 0 ≤ i ∧ i < pq..length ∧ 0 ≤ j ∧
                  j < pq..length ∧ (j = 2*i + 1 ∨ j = 2*i + 2) →
                  pq..queue.[i]..pr ≥ pq..queue.[j]..pr)) ∧
              theinv CardInv ∧ theinv CapacityInv ∧
              theinv InitialLength ∧ theinv LengthInv ∧
              theinv NonNullInv ∧ theinv DistinctInv ∧
              theinv NullInv ∧ theinv HiddenInv ∧ theinv InjInv ∧
              theinv ContentsInj"
    modifies arrayState
    ensures "(∀pq. pq..nodes = old (pq..nodes)) ∧
             (∀a i. a ≠ queue → a.[i] = old (a.[i])) ∧
             alloc = old alloc ∧ theinvs" */
{ ... }

private void inductProof()
/*: requires "init ∧ 0 < length ∧ theinvs"
    ensures "(∀k. 0 ≤ k ∧ k < length → queue.[0]..pr ≥ queue.[k]..pr) ∧
             theinvs" */
{ ... }
```

Figure 5-4: Private Methods of the PriorityQueue Class (continued from Figure 5-3)

binary heap property after the removal of a node). The requires clause indicates that right() must be invoked on an index that is greater than or equal to zero. The ensures clause indicates that right() returns an index that is greater than i and corresponds to the right child of the node at i. It also indicates that right() does not allocate any objects. There is no modifies clause because the right() method does not change the program state.

5.5.4 The contains() Method

The contains() method is a private instance method that takes an object o1 and a priority p1, and returns a boolean value corresponding to whether the pair (o1, p1) is in the priority queue. It is used by the add() method, to determine whether an object-priority pair being added is already in the queue. The requires clause indicates that contains must be invoked on an initialized priority queue, that o1 must be non-null, and that the invariants of the priority queue (theinvs) must hold. The ensures clause indicates that contains returns true if (o1, p1) is in the queue, and false otherwise. It also indicates that contains() preserves the invariants of the priority queue.

5.5.5 The resize() Method

The resize() method is a private instance method that increases the capacity of the priority queue. It is used by the add() method to increase the size of the array queue to accommodate a new entry beyond the current capacity of the queue. The requires clause indicates that resize() must be invoked on an initialized priority queue and that the priority queue invariants must hold. Unlike public methods, for which invariants are implicitly conjoined to both preconditions and postconditions, private methods are not required to either depend on, or restore invariants, since the data structure may be in an inconsistent state as it is being modified by private methods.

The modifies clause indicates that resize() may modify the queue field and the abstract state component capacity of the given priority queue. It also indicates that resize() may modify the contents of arrays. The queue field is included in the modifies clause because resize() is a private method; the modifies clause of a private method must include all state that may be modified, while that of a public method need only include public state that may be modified. The ensures clause indicates that the effect of invoking resize() is to increase the capacity of the queue such that it is greater than the number of objects currently in the queue. It also indicates that resize() preserves the contents and init components of all PriorityQueue objects, that the only array that the method may modify is the queue array of the given priority queue, and that the invariants of the priority queue are restored when the method returns.

Jahob automatically verifies frame conditions for unmodified dependent variables, such as contents and init, but the system does not automatically include such frame conditions as part of the instantiated postcondition for calls when the variables are defined in terms of state that is modified. The explicit frame conditions in the post-condition is therefore used for verifying methods that call resize(). The postcondition of resize() also includes an explicit frame condition for arrays. This is needed to verify

the frame condition for add(), whose method contract allows for the modification of arrays only if they are in hidden.

5.5.6 The addOnly() Method

The addOnly() method is a private instance method that adds an object to the priority queue, given that the object-priority pair being added is not already in the queue. It is used by the add() method to perform the actual addition into the queue. The requires clause indicates that the method must be invoked on an initialized priority queue, that the object o1 being added must be non-null, and that the object-priority pair (o1, p1) being added must not already be in the queue. It also requires that the capacity of the queue be greater than the number of objects already in the queue, and that the priority queue invariants hold.

The modifies clause indicates that the method may modify the abstract state component contents and the length field of the given priority queue. It also indicates that the method may modify the contents of arrays, and the values of the Node.ob and Node.pr fields. The ensures clause indicates that the effect of addOnly() is to add the pair (o1, p1) to the contents of the queue and that the length field of the given queue increases by one. It also indicates that the abstract component contents is unchanged for priority queues other than the receiver object, that the only array modified is the queue array of the given priority queue, and that the priority queue invariants are restored at the end of the method.

The purpose of having a separate addOnly() method, as opposed to implementing its behavior directly in add(), is to simplify the verification task. Methods that directly examine or modify the concrete state of the data structure are more difficult to verify, as are larger methods. In this case, the functionality of add() is divided between contains(), resize(), and addOnly().

5.5.7 The heapify() Method

The heapify() method is a private instance method that recursively restores the binary heap property on a priority queue for which the property holds for all but one node. It is used by the poll() method to restore the binary heap property after the root node has been removed and replaced with the right-most leaf node. The requires clause indicates that heapify() must be invoked on an initialized priority queue and that the parameter i (corresponding to the node for which the binary heap property does not hold) must be in the range [0, length) of the nodes in use by the queue. The conjunct with label GlobalOrderingPre indicates that the binary heap property must hold for all nodes in the queue other than the node at index i. The conjunct with label LocalOrderingPre indicates that the parent of the node at i must be in the correct position in the heap relative to the children of the node at i. The conjunct with label OrderedFrame indicates that the binary heap property holds for all PriorityQueue objects other than the receiver. The requires clause also indicates that all other invariants of the priority queue must hold.

The modifies clause indicates that the method may modify the contents of arrays. The ensures clause indicates that heapify() preserves the abstract state component nodes for all objects and that the only array it modifies is the queue array of the given priority queue. It also indicates that heapify() does not allocate any objects and that the priority queue invariants hold at the end of the method.

Although the size of the heapify() precondition is somewhat daunting, most of the conjuncts are simply named invariants of the class. The main source of complexity comes from specifying the partially-violated ordering invariant. The explicit statement of the relevant properties in the precondition is nevertheless useful for precisely documenting how the ordering invariant is violated and the parts of the state for which it is maintained. Fortunately, this precondition occurs in a private method, so clients of the data structure are not affected.

5.5.8 The inductProof() Method

The inductProof() method is a proof method. It is private instance method that contains no Java statements, only proof commands. It inductively proves that the root node of the priority queue corresponds to an element in the queue with the maximal priority and is called by both the peek() and poll() methods. Its requires clause indicates that the proof in inductProof() applies to an initialized priority queue with length greater than zero, given that all the priority queue invariants hold. The ensures clause indicates what the proof guarantees. Specifically, inductProof() shows that the root node corresponds to an element in the queue with the maximal priority. It also indicates that the method preserves the priority queue invariants. There is no modifies clause because the method does not change the program state.

Encapsulating the proof in a method not only makes the code easier to read, it also makes it possible to reuse the same proof (in other contexts where the precondition of the proof method holds) by simply invoking the proof method. The inductProof() method is invoked by both the peek() and poll() methods to establish that the zeroth element of the queue array holds an element with the maximal priority. Note that inductProof() has the receiver as an implicit parameter, so that the proof is not only encapsulated but also parameterized.

5.6 Implementation and Verification

This section illustrates the implementation and verification of the PriorityQueue class using the example of the peek() and inductProof() methods.

5.6.1 The peek() Method

Figure 5-5 presents the body of the peek() method. The peek() method returns an object from the priority queue with the maximal priority, or null if the queue is empty. It does this by first testing the length field of the priority queue. If length is zero, then the queue is empty, and the method returns null. Otherwise, peek() returns the ob

```
1  public Object peek()
2  /*: requires "init"
3      ensures "(contents = ∅ → result = null) ∧
4               (contents ≠ ∅ →
5                 (∃p.(result,p) ∈ contents ∧
6                   (∀x y.(x,y) ∈ contents → p ≥ y)))" */
7  {
8      if (length == 0) return null;
9
10     inductProof();
11
12     /*: note InContents: "(queue.[0]..ob,queue.[0]..pr) ∈ contents"
13         from ProcedurePrecondition, FalseBranch, thisType, nodes_def,
14         contents_def; */
15     /*: witness "queue.[0]..pr" for
16         PostCond: "∃p.(queue.[0]..ob,p) ∈ contents ∧
17                       (∀x y.(x,y) ∈ contents → p ≥ y)"; */
18     return queue[0].ob;
19 }
```

Figure 5-5: PriorityQueue Proof Language Example

field of the zeroth element of queue. This element is guaranteed by the binary heap property to hold an object in the queue with the maximal priority. (Ignore, for the moment, the inductProof() method, which we will discuss shortly.)

To verify peek(), Jahob must ensure that its postcondition holds. Unfortunately, the provers are unable to automatically establish the postcondition for the case where the method returns an object associated with the maximal priority. In part this is because the relevant portion of the postcondition contains an existentially quantified formula, and many of the provers have difficulty proving this type of formula. But the more serious problem is that to show that the zeroth element of queue has the maximal priority requires an inductive proof, which the automated provers are unable to perform. In the absence of developer guidance, the provers are therefore unable to verify the peek() method.

The Witness Command

We solve this problem using the witness and note commands from Jahob's integrated proof language. The witness command in line 15 addresses the existential quantifier in the postcondition, by identifying queue.[0]..pr as satisfying the existentially quantified portion of the postcondition. This command directs Jahob to prove that queue.[0]..pr is indeed the witness—i.e. that $(\mathsf{queue.[0]..ob}, \mathsf{queue.[0]..pr}) \in \mathsf{contents} \wedge (\forall \mathsf{xy}.(\mathsf{x},\mathsf{y}) \in \mathsf{contents} \rightarrow \mathsf{p} \geq \mathsf{y})$ holds.

The Note Command

By itself, the witness command is not sufficient for enabling the provers to establish the desired postcondition. One reason for this is that irrelevant assumptions in the

verification conditions create such a large search space that the provers may fail to successfully explore it in a reasonable amount of time. The optional from clause in note commands addresses this problem, by identifying the relevant assumptions. The note command in line 12 directs Jahob to prove an intermediate lemma and to label the proved lemma InContents. This lemma captures the first of two conjuncts in the property Jahob needs to prove to establish that queue.[0]..pr is an appropriate witness for the existentially quantified portion of the postcondition. The from clause in this note command eliminates irrelevant assumptions by identifying the properties that the provers need to use to establish InContents—specifically, 1) the precondition of the method, 2) the negation of the branch condition, 3) the type of the receiver object and the fact that it is allocated, and 4) the definitions of the specification variables nodes and contents. All of these facts are properties that Jahob makes available by default through standard labels. With this guidance, the theorem provers easily establish the InContents lemma, which is then added to the assumption base and available for subsequent reasoning.

Call to inductProof

The second conjunct that Jahob needs to prove requires an inductive proof to establish that the zeroth element of queue has the maximal priority. This is necessary because the priority queue ordering property, OrderedInv, is a local property that establishes the ordering between a parent and child node in the heap. The property Jahob needs to establish—that the root node has the maximal priority—is a global property of the queue that follows inductively from OrderedInv.[4]

The proof of this property is encapsulated in the inductProof() instance method. The peek() method calls inductProof() using a standard Java method call, which instantiates the postcondition of inductProof() in peek() for the receiver object. This postcondition provides the necessary guidance that the provers need to prove the second conjunct of the pickWitness command. The pickWitness command is then able to establish the existentially quantified portion of the postcondition. From that result, the provers are able to prove that the postcondition of peek() holds.

5.6.2 The inductProof() Method

Figure 5-6 presents the body of the inductProof() method. It contains a very detailed proof, written in Jahob's proof language, which is also the longest proof of a single property in our set of benchmark programs. The method contract for inductProof() captures the precondition and postcondition for the proof. The requires clause indicates that the proof applies to an initialized priority queue object with length greater than zero, provided that all of the priority queue data structure invariants hold. The

[4]It is also possible to specify the priority queue with the global ordering property as an invariant, but the local property would still be necessary for inductively re-establishing the global property whenever objects are added to the heap, effectively moving the inductive proof from the poll() method to the add() method.

```
1  private void inductProof()
2  /*: requires "init ∧ 0 < length ∧ theinvs"
3      ensures "(∀k.0 ≤ k ∧ k < length →
4                  queue.[0]..pr ≥ queue.[k]..pr) ∧ theinvs"  */
5  {
6    {
7      /*: induct InGeneral: "∀x.x ≤ z ∧ 0 ≤ x ∧ x < length →
8          queue.[0]..pr ≥ queue.[x]..pr" over z::int; */
9      {
10       /*: assuming InductHyp: "∀x.x ≤ z ∧ 0 ≤ x ∧ x < length →
11           queue.[0]..pr ≥ queue.[x]..pr"; */
12       {
13         //: pickAny x::int suchThat "x ≤ z+1 ∧ 0 ≤ x ∧ x < length";
14         {
15           //: assuming EqHyp: "x = z + 1";
16           {
17             //: assuming OddHyp: "x mod 2 = 1";
18             //: note OddParent: "∃ y. y + y + 1 = x" from OddHyp;
19             {
20               //: pickWitness oddp::int suchThat "oddp + oddp + 1 = x";
21               //: note ParentGe: "queue.[oddp]..pr ≥ queue.[x]..pr";
22               //: note ParentInduct: "queue.[0]..pr ≥ queue.[oddp]..pr";
23               //: note OddGe: "queue.[0]..pr ≥ queue.[x]..pr";
24             }
25             //: note OddCase: "queue.[0]..pr ≥ queue.[x]..pr";
26           }
27           {
28             //: assuming EvenHyp: "x mod 2 = 0";
29             ...
30             //: note EvenCase: "queue.[0]..pr ≥ queue.[x]..pr";
31           }
32           /*: note EqConc: "queue.[0]..pr ≥ queue.[x]..pr"
33               from EvenCase, OddCase; */
34         }
35         //: note InductConc: "queue.[0]..pr ≥ queue.[x]..pr" forSuch x;
36       }
37       /*: note InductCase: "∀x.x ≤ z+1 ∧ 0 ≤ x ∧ x < length →
38           queue.[0]..pr ≥ queue.[x]..pr"; */
39     }
40   }
41   /*: note MaxInQueue:
42       "∀k.0 ≤ k ∧ k < length → queue.[0]..pr ≥ queue.[k]..pr"
43       from InGeneral; */
44 }
```

Figure 5-6: PriorityQueue Proof Method Example

ensures clause specifies what the method proves—that the zeroth element of queue has the maximal priority, and that the priority queue invariants are preserved.

Applying Induction

The proof begins with an induct block which instructs Jahob to apply induction to the formula $\forall x.x \leq z \wedge 0 \leq x \wedge x < \mathsf{length} \rightarrow \mathsf{queue}.[0]..\mathsf{pr} \geq \mathsf{queue}.[x]..\mathsf{pr}$ over z by proving the base and inductive cases of the induction. The base case of the induction is $\forall x.x \leq 0 \wedge 0 \leq x \wedge x < \mathsf{length} \rightarrow \mathsf{queue}.[0]..\mathsf{pr} \geq \mathsf{queue}.[x]..\mathsf{pr}$. The automated provers discharge this case automatically, requiring no proof commands. But the provers are unable to discharge the inductive case automatically. The proof commands that follow provide the guidance necessary for the provers to successfully prove this case.

The Inductive Case

The inductive case is given by the following formula:

$$(\forall x.x \leq z \wedge 0 \leq x \wedge x < \mathsf{length} \rightarrow \mathsf{queue}.[0]..\mathsf{pr} \geq \mathsf{queue}.[x]..\mathsf{pr}) \rightarrow$$
$$(\forall x.x \leq z + 1 \wedge 0 \leq x \wedge x < \mathsf{length} \rightarrow \mathsf{queue}.[0]..\mathsf{pr} \geq \mathsf{queue}.[x]..\mathsf{pr})$$

Its proof begins with an assuming command. This command creates a hypothetical block where the inductive hypothesis InductHyp (the left hand side of the above implication) is true. The proof commands that follow work within this context to prove the right-hand side of the implication.

The PickAny Command

The first command in this block is a pickAny command that selects an arbitrary x for which $x \leq z + 1 \wedge 0 \leq x \wedge x < \mathsf{length}$. The task now is to prove that $\mathsf{queue}.[0]..\mathsf{pr} \geq \mathsf{queue}.[x]..\mathsf{pr}$. There are two cases to consider. In the first case, $x < z + 1$. This case follows directly from the inductive hypothesis. The provers are able to prove it automatically without the need for any proof commands. In the second case, $x = z+1$. The following proof commands provide the necessary guidance for the provers to prove this case.

The Odd Case

First, an assuming command with the label OddHyp creates a hypothetical block for the case when x is odd—i.e. when $x \bmod 2 = 1$. A note command with a from clause instructs Jahob to prove the intermediate assertion OddParent using only the hypothesis OddHyp. OddParent indicates that there exists an index corresponding to the parent of the node at index x. A pickWitness command names this index oddp. A series of note commands follow. The first note command, at line 21, directs Jahob to establish that the priority of the (parent) node at index oddp is greater than or equal to the priority of the (child) node at index x, and to label the proved lemma ParentGe. This property follows from the local ordering invariant OrderedInv. The second note

command, at line 22, directs Jahob to prove that the priority of the (root) node at index 0 is greater than or equal to the priority of the node at index oddp, and to label the proved lemma ParentInduct. This property follows from the inductive hypothesis. Finally, the third note, at line 23, directs Jahob to prove the desired result—that the priority of the root node is greater than or equal to the priority of the node at x—and to label the proved lemma OddGe. This property follows from the previous two note commands. In each of these note commands, the provers are able to establish the result without a from clause specifying which facts to use. Finally, the note command at line 25 (labeled OddCase) concludes the proof by closing the OddHyp assuming block.

The Even Case

The proof for an even index x proceeds in a similar way, and results in the proved lemma EvenCase. The note command at line 32 combines the even and odd cases to conclude the EqHyp assuming block. The note commands at line 35 and 37 close the enclosing pickAny and assuming blocks, respectively, to conclude the proof of the inductive case. The result is the successful proof of the following property, which has the label InGeneral from the induct command at line 7:

$$\forall z.0 \leq z \rightarrow (\forall x.x \leq z \wedge 0 \leq x \wedge x < \mathsf{length} \rightarrow \mathsf{queue}.[0]..\mathsf{pr} \geq \mathsf{queue}.[x]..\mathsf{pr})$$

A final note command at line 41 with label MaxInQueue directs Jahob to use only the proved lemma InGeneral to conclude that the zeroth element of the queue has the maximal priority—i.e., that $\forall k.0 \leq k \wedge k < \mathsf{length} \rightarrow \mathsf{queue}.[0]..\mathsf{pr} \geq \mathsf{queue}.[k]..\mathsf{pr}$. This is the property of interest in the postcondition of inductProof(), which is now proved.

5.7 Discussion

The priority queue is an unusual data structure in that it contains an implicit tree structure, encoded within an array. In that sense, it is both an array-based data structure and a recursive data structure. It was one of the more challenging data structures to verify, due to the combination of structural and ordering invariants maintained by the data structure, and the use of layered abstraction. Data structures of similar complexity, such as the hash table and binary search tree, were similarly challenging to verify.

The verification effort centered on: 1) properties involving the abstract component content, 2) the cardinality invariant CardInv, and 3) the inductive proof showing that the zeroth element of queue corresponds to the maximal priority. Properties involving content include frame conditions showing that content did not change for other priority queue objects, and postconditions describing the updated state for the given priority queue. Both types of properties involved proving equivalence between sets. It was often necessary to show this equivalence by considering elements in one set and proving inclusion in the other set, and vice versa. This pattern occurs sufficiently often in our verified data structures that it may be beneficial to include a translation

for set equivalences within Jahob to lessen the proof burden on the developer. It was also common to use case analysis in proof commands involving content, to direct the provers to consider the case of the receiver object separately from all other objects. We also used this pattern for the verification of CardInv. Verification for the cardinality invariant was, in general, straightforward. The proof commands involved identifying the different cases, and introducing the relevant intermediate lemmas that the simple cardinality prover needed to re-establish the invariant.

The inductive proof for the priority queue is the only inductive proof in the data structures we verified. It is also one of the more interesting and involved proofs, due to properties of the implicit tree structure (such as the need to consider even and odd cases). The large number of proof commands needed is partly due to the large number of arithmetic properties involved—both for the array indices, and for the ordering of priorities. In general, the first-order provers E and SPASS are adept at handling properties involving universal quantifiers, but are typically unable to handle arithmetic properties. The reverse is true for the SMT provers and for the Isabelle proof script. The net effect is that developer guidance is often needed to coordinate the efforts of multiple provers when the verification conditions involve both universal quantification and arithmetic properties, as in the priority queue. Since all of the data structures we verified involve some universally quantified properties, the data structures that use proof commands most extensively are those that also maintain arithmetic invariants.

5.8 Summary

In this chapter, we show how we use Jahob to specify and verify a PriorityQueue class that implements a priority queue data structure. Our priority queue is implemented using a binary heap stored in an array. The abstract state of the queue is represented using a set of object-priority pairs, corresponding to the contents of the queue. The specification includes data structure invariants that express constraints on the concrete and abstract state of the queue. These include constraints on the ordering of the elements in the queue, injectivity invariants that express the lack of sharing between different priority queues and different parts of the same queue, as well as constraints on the values of certain fields. The invariants are necessary for verifying important properties of the data structure, including method contracts, the invariants themselves, and the lack of null dereferences and array bounds violations in the implementation. The PriorityQueue class exports a number of different public operations on the queue, including add(), peek(), and poll(), which add, examine, and remove elements from the queue, respectively. These operations are supported by a number of private methods, which are also verified. We show, in detail, how we verified the peek() method using Jahob's integrated reasoning system and the commands of the proof language. Part of this proof is implemented as a proof method which encapsulates a parameterized, inductive proof, making it possible to soundly instantiate the same proof in two different contexts within the class.

Although the verification of the priority queue is unusual in that it involved an

interesting inductive property, in other respects, it is similar to other complex data structures that we have verified. We used proof commands to guide the provers in considering case splits (between the receiver object and other objects), in proving set equivalences, and in decomposing the proof into the appropriate intermediate lemmas. In particular, the combination of universally quantified properties and arithmetic properties in the verification resulted in the need for proof commands to coordinate the efforts of different provers to handle the two different types of properties. In our experience, provers adept at handling universal quantifiers are generally weak in proving properties involving arithmetic, and vice versa. As data structures that contain an unbounded number of objects often involve universally quantified properties, those data structures that also maintain arithmetic invariants, like the priority queue, tend to require a substantial number of proof commands to verify.

Chapter 6

Association List

In this chapter, we describe an association list data structure which we have verified using Jahob. The AssociationList class implements a map using a singly-linked list. In terms of implementation, it is one of the simpler data structures that we have verified. However, the invariants that it maintains, which enable it to implement a map interface, are non-trivial. We use the association list to illustrate the specification and verification of a recursive data structure with non-trivial properties, that nevertheless verifies without the use of any proof commands.

6.1 The Concrete State

Figure 6-1 presents part of the AssociationList class and a Node class which stores the key-value pairs in the list. The concrete state of the association list consists of a singly-linked list of Node objects, with the private instance field first referring to the first node in the linked list. Each Node object consists of a key field storing the key for a mapping in the association list, a value field storing the corresponding value, and a next field storing the next Node object in the linked list. The claimedby annotation in the declaration of the Node class indicates that only methods in the AssociationList class may modify the fields and specification variables of Node objects.

6.2 The Abstract State

Unlike the priority queue data structure, whose abstract state is represented using only dependent variables, the abstract state of the association list is represented using a combination of ghost variables and dependent specification variables. The specification variables in the priority queue consist of the dependent variable contents, representing the set of key-value pairs stored in an association list, and the ghost variable con, representing the set of key-value pairs stored in a linked list starting at a given Node object. The ghost variable con allows us to use recursion to encode properties that depend on reachability.[1] The initial value of con is set to the empty

[1] Alternatively, we could specify con as a dependent variable using the transitive closure operator. In that case, we would need to use the MONA decision procedure, which is able to handle properties

```
public /*: claimedby AssociationList */ class Node {
    public Object key;
    public Object value;
    public Node next;
    //: public ghost specvar con :: "(obj * obj) set" = "∅"
}

public class AssociationList {
    private Node first;
    /*:
      public specvar contents :: "(obj * obj) set";
      vardefs "contents == first..con";

      static specvar edge :: "obj ⇒ obj ⇒ bool";
      vardefs "edge == (λx y. (x ∈ Node ∧ y = x..next) ∨
                        (x ∈ AssociationList ∧ y = x..first))";

      invariant ConDef: "∀x. x ∈ Node ∧ x ∈ alloc ∧ x ≠ null →
        x..con = {(x..key, x..value)} ∪ x..next..con ∧
        (∀v. (x..key, v) ∉ x..next..con)";
      invariant ConNull: "null..con = ∅";
      invariant ConNonNull:
      "∀z x y. z ∈ Node ∧ z ∈ alloc ∧ (x, y) ∈ z..con →
         x ≠ null ∧ y ≠ null";
      invariant ConAlloc:
      "∀z x y. z ∈ Node ∧ z ∈ alloc ∧ (x, y) ∈ z..con →
         x ∈ alloc ∧ y ∈ alloc";
      invariant InjInv:
      "∀x1 x2 y. y ≠ null ∧ edge x1 y ∧ edge x2 y → x1 = x2";
      invariant MapInv:
      "∀k v0 v1. (k, v0) ∈ contents ∧ (k, v1) ∈ contents → v0 = v1";
      invariant NonNullInv:
      "∀k v. (k, v) ∈ contents → k ≠ null ∧ v ≠ null"; */
    ...
}
```

Figure 6-1: AssociationList Example

set to ensure that the priority queue invariants hold in the initial state.

The specification also declares a dependent specification variable edge, which is a static specification variable that serves as a shorthand for a lambda expression. The result of applying this lambda expression to an object is a boolean value that denotes whether: 1) a next edge occurs between two Node objects, or 2) a first edge occurs between an AssociationList object and a Node object. The use of edge in the invariants of the AssociationList class highlights how specification variables that serve as shorthands can enable specifications that are more concise and easier to understand.

6.3 The Abstraction Function and Invariants

Since the abstract state of the association list consists of both ghost variables and dependent variables, the abstraction function is given by a combination of vardefs declarations and data structure invariants. The vardefs declarations provide components of the abstraction function pertaining to dependent variables, while data structure invariants provide the components pertaining to ghost variables, as well as additional constraints on the concrete and abstract state. As such, the vardefs declaration for contents indicates that the contents of the association list is given by first..con, the contents of the linked list starting with the first node.

The invariants ConNull and ConDef give the definition for the abstract state component con, using recursion to state a reachability property. The ConNull invariant, which gives the base case, indicates that the value of con for null is the empty set. The ConDef invariant, which gives the recursive case, indicates that, for an allocated, non-null, Node object, con consists of the key-value pair stored at that node, plus the value of con for the next node in the linked list. The ConDef invariant additionally states that next..con does not contain a value for the key stored at the current node, thus ensuring that a key occurs at most once in the list.

In addition to the invariants that comprise the abstraction function, the AssociationList class contains the following invariants that specify the valid concrete and abstract state of the association list data structure:

- The ConNonNull invariant indicates that every key and value in a pair in the set con are non-null. This ensures that only non-null keys and values are stored in nodes.

- The ConAlloc invariant indicates that every key and value in a pair in con are allocated. This invariant makes it possible to apply other invariants in the class to a Node object. As described in previous chapters, invariants that contain implicit references to the receiver are automatically universally quantified over all allocated objects of the class. Most of the time, the objects used in the

involving transitive closure. Here we demonstrate how to encode transitive closure as a recursive property using ghost variables. This approach enables the use of first-order provers and other provers not able to handle properties that contain the transitive closure operator.

verification are known to be allocated, since Jahob automatically provides the allocation information for concrete objects in the implementation. But because con is user-defined, the system is not aware of any particular constraints on the objects in the set, making it necessary to state this invariant explicitly.

- The InjInv invariant expresses an injectivity property. It indicates that every node is either pointed to by a single first edge from an AssociationList object or by a single next edge from another Node object. Here we use the dependent variable edge as a shorthand, to state that Node objects are neither shared among different association lists, nor among different parts of the same list (i.e. the list is acyclic).

- The MapInv invariant indicates that the association list contains no more than one mapping for every key.

- The NonNullInv invariant that every key and value mapped in the association list are non-null.

Although the last two invariants follow from other invariants in the class, they are declared separately to simplify the verification. Specifically, declaring these properties separately as invariants avoids the need for re-establishing the properties every time they are needed in the verification. Instead, they need only be re-established when the invariant is violated by changes in the underlying state. In general, invariants that follow from other invariants can be useful both for documentation purposes and for easing verification.

6.4 Method Contracts for Public Methods

Figure 6-2 presents the method contracts for all the public methods of the AssociationList class. These methods constitute the interface that the AssociationList class exports to its clients. The interfaces for the AssociationList constructor and the containsKey(), get(), isEmpty(), put(), and remove() methods are modeled after the interfaces for the corresponding methods in the java.util.Map interface. The methods add() and replace() are methods not found in java.util.Map, but we include these methods because they offer alternative interfaces that clients may find useful.

6.4.1 The AssociationList Constructor

The AssociationList constructor is a public constructor that creates a new, empty, initialized AssociationList object. The modifies clause indicates that it may modify the contents component of the association list under construction. The ensures clause indicates that the resulting association list is empty. The lack of a requires clause indicates that the constructor has no precondition.

```
public AssociationList()
/*: modifies contents
    ensures "contents = ∅" */
{ ... }

public boolean containsKey(Object k0)
//: ensures "result = (∃v. ((k0, v) ∈ contents))"
{ ... }

public Object get(Object k0)
/*: requires "k0 ≠ null"
    ensures "((result ≠ null) → ((k0, result) ∈ contents)) ∧
             ((result = null) → (¬(∃v. (k0, v) ∈ contents)))" */
{ ... }

public boolean isEmpty()
//: ensures "result = (contents = ∅)"
{ ... }

public Object put(Object k0, Object v0)
/*: requires "k0 ≠ null ∧ v0 ≠ null"
    modifies contents
    ensures "contents = old contents − {(k0, result)} ∪ {(k0, v0)} ∧
             (result = null → ¬(∃v. (k0, v) ∈ old contents)) ∧
             (result ≠ null → (k0, result) ∈ old contents)" */
{ ... }

public Object remove(Object k0)
/*: requires "k0 ≠ null ∧ (∃v. (k0, v) ∈ contents)"
    modifies contents
    ensures "contents = old contents − {(k0, result)} ∧
             (result = null → ¬(∃v. (k0, v) ∈ contents)) ∧
             (result ≠ null → (k0, result) ∈ old contents)" */
{ ... }

public void add(Object k0, Object v0)
/*: requires "k0 ≠ null ∧ v0 ≠ null ∧ ¬(∃v. (k0, v) ∈ contents)"
    modifies contents
    ensures "contents = old contents ∪ {(k0, v0)}" */
{ ... }

public Object replace(Object k0, Object v0)
/*: requires "k0 ≠ null ∧ v0 ≠ null ∧ (∃ v. (k0, v) ∈ contents)"
    modifies contents
    ensures "contents = old contents − {(k0, result)} ∪ {(k0, v0)} ∧
             (k0, result) ∈ old contents" */
{ ... }
```

Figure 6-2: Exported Operations of the AssociationList Class

6.4.2 The containsKey() Method

The containsKey() method is a public instance method that takes a key k0 and returns a boolean value indicating whether k0 has a mapping in the association list. The ensures clause indicates that the method returns true if k0 corresponds to a mapping in the association list, and false otherwise. There is no requires or modifies clause as containsKey() has no preconditions and does not modify the program state.

6.4.3 The get() Method

The get() method is a public instance method that takes a key k0 and returns the value to which k0 is mapped in the association list. The requires clause indicates that k0 must be non-null. The ensures clause indicates that if get() returns null, then the association list does not contain a mapping for k0. But if get() returns a non-null value, then the return value is the value to which k0 is mapped in the association list. There is no modifies clause since get() does not modify the program state.

6.4.4 The isEmpty() Method

The isEmpty() method is a public instance method that returns a boolean value indicating whether the given association list is empty. The ensures clause indicates that the method returns true if the association list is empty, and false otherwise. There is no requires or modifies clause because isEmpty() has no precondition and does not modify the program state.

6.4.5 The put() Method

The put() method is a public instance method that adds a mapping to the association list. The requires clause for put() indicates that the key k0 and value v0 being added must be non-null. The modifies clause indicates that put() may modify the contents of the given association list. The ensures clause indicates that the value of contents after invoking put() is the result of removing the pair (k0, result) from contents, and adding the pair (k0, v0). It also indicates that if the return value is null, then the association list did not originally contain a mapping for the key k0. But if the return value is not null, then it is the value to which k0 was originally mapped.

6.4.6 The remove() Method

The remove() method is a public instance method that removes a mapping from the association list. The requires clause indicates that the key k0 being removed must be non-null. The modifies clause indicates that remove() may modify the contents of the given association list. The ensures clause indicates that the value of contents after invoking remove() is the result of removing the pair (k0, result) from contents. As is the case for put(), the ensures clause also indicates that if the return value is null, then the association list did not originally contain a mapping for k0. But if the return value is not null, then it is the value to which k0 was originally mapped.

```
private boolean _containsKey(Object k0)
/*: requires "theinvs"
    ensures "result = (∃v. ((k0, v) ∈ contents)) ∧ theinvs" */
{ ... }

private void _add(Object k0, Object v0)
/*: requires "k0 ≠ null ∧ v0 ≠ null ∧ ¬(∃v. (k0, v) ∈ contents) ∧
            theinvs"
    modifies contents, first, "new..key", "new..value", "new..next",
            "new..con"
    ensures "contents = old contents ∪ {(k0, v0)} ∧ theinvs" */
{ ... }

private Object _remove(Object k0)
/*: requires "k0 ≠ null ∧ (∃v. (k0, v) ∈ contents) ∧ theinvs"
    modifies contents, first, next, con
    ensures "contents = old contents - {(k0, result)} ∧
            (k0, result) ∈ old contents ∧ theinvs" */
{ ... }
```

Figure 6-3: Private Methods of the AssociationList Class

6.4.7 The add() Method

The add() method is a public instance method that adds a mapping to the association list. It is similar to the put() method, but requires that the association list contain no previous mapping for the given key. It is more efficient than put() and offers an alternative interface for adding a mapping to the association list when its precondition is known to hold. The requires clause additionally indicates that the key k0 and value v0 being added must be non-null. The modifies clause indicates that it may modify the contents of the given association list. The ensures clause indicates that it adds a mapping for the given key-value pair to contents.

6.4.8 The replace() Method

The replace() method is a public instance method that maps an existing key in the association list to a new value. It is similar to the put() method, but requires that the association list already contain a mapping for the given key. It is more efficient than put() and offers an alternative interface for adding a binding to the association list when its precondition is known to hold. The requires clause additionally indicates that the key k0 and value v0 being added must be non-null. The modifies clause indicates that replace() may modify the contents of the association list. The ensures clause indicates that replace() removes the previous mapping for the key from contents, adds a mapping for the given key-value pair, and returns the previously bound value.

6.5 Method Contracts for Private Methods

Figure 6-3 presents the method contracts for the private methods of the AssociationList class. These methods implement various functionalities that are needed to support the public operations of the association list. Notice that the contracts for these methods, and in particular, the modifies clauses, refer to private fields and specification variables (such as con, first, etc.) not mentioned in the contracts for the public methods of the class. Although the contracts for the public methods are expressed only in terms of the public state, the constraints between the public and private state of the data structure are such that these contracts comprise a complete functional specification of the behavior of the data structure, from the perspective of the association list client. But for private methods, whose clients are other methods in the AssociationList class, the contracts must also include references to the private state to be comprehensive.

6.5.1 The _containsKey() Method

The _containsKey() method is a private instance method that takes a key k0 and returns a boolean value indicating whether k0 has a mapping in the association list. It implements the functionality needed by the public containsKey() method, and is also invoked by the put() and remove() methods. The requires clause indicates that _containsKey() must be invoked on a non-null key k0, and that the invariants of the association list must hold. The ensures clause indicates that _containsKey() returns true if the association list contains a mapping for k0, and false otherwise. It also indicates that _containsKey() preserves the association list invariants. There is no modifies clause because _containsKey() does not modify the program state.

6.5.2 The _add() Method

The _add() method is a private instance method that adds a mapping to the association list for a key k0 that is not already mapped. It implements the functionality of the add() method and is also invoked by put() and replace(). The requires clause of _add() indicates that it must be invoked on a key k0 and a value v0 that are non-null, that the association list must not already contain a mapping for k0, and that the invariants of the association list hold. The modifies clause indicates that _add() may modify the abstract component contents and the first field of the given association list, as well as the key, value, next, and con components of objects that it allocates. The ensures clause indicates that the effect of invoking _add() is to add the pair (k0, v0) to the contents of the association list. It also indicates that _add() re-establishes the invariants of the association list at the end of the method.

Although the modifies clause of the _add() method lists a number of different components, including the first field, and the key, value, next, and con components of objects allocated by add(), as being modified, the effect of the modification on these state components is not given in the ensures clause. Despite this, the contract is comprehensive from the perspective of the methods that call _add(). The resulting method contract concisely captures all the relevant functional properties of _add(),

```
1  public Object get(Object k0)
2  /*: requires "k0 ≠ null"
3      ensures "((result ≠ null) → ((k0, result) ∈ contents)) ∧
4               ((result = null) → (¬(∃ v. (k0, v) ∈ contents)))" */
5  {
6      Node current = first;
7      while //: inv "∀v. ((k0, v) ∈ contents) = ((k0, v) ∈ current..con)"
8          (current != null) {
9
10         if (current.key == k0)
11             return current.value;
12
13             current = current.next;
14     }
15     return null;
16 }
```

Figure 6-4: Body of AssociationList.get()

while providing a form of abstraction by not exposing all the concrete details of the implementation.

6.5.3 The _remove() Method

The _remove() method is a private instance method that takes a key k0 that is known to be in the association list and removes the corresponding mapping. It is invoked by the remove(), put(), and replace() methods. The requires clause indicates that _remove() must be invoked on a key k0 that is non-null and has a corresponding value in contents, and that the invariants of the association list must hold. The modifies clause indicates that the _remove() method may modify the abstract component contents and the first field of the association list, and the abstract component Node.con and the Node.next field. The ensures clause indicates that the effect of invoking _remove() is to remove the pair (k0, result) from contents and that this pair was originally in contents. It also indicates that _remove() re-establishes the invariants of the association list at the end of the method.

6.6 Implementation and Verification

This section illustrates the implementation and verification of the AssociationList class using the example of the get() and remove() methods. It describes the implementation and verification of these methods and of all the methods in the AssociationList class that they invoke either directly or transitively. All the methods in the AssociationList class verify automatically without guidance from any proof commands.

```
1  public Object remove(Object k0)
2  /*: requires "k0 ≠ null ∧ (∃v. (k0, v) ∈ contents)"
3      modifies contents
4      ensures "contents = old contents - {(k0, result)} ∧
5               (result = null → ¬(∃v. (k0, v) ∈ contents)) ∧
6               (result ≠ null → (k0, result) ∈ old contents)" */
7  {
8      if (_containsKey(k0))
9          return _remove(k0);
10     else
11         return null;
12 }
```

Figure 6-5: Body of AssociationList.remove()

6.6.1 The get() Method

Figure 6-4 presents the body of the get() method. The get() method takes a key k0, and uses a while loop to search through the nodes of the association list for this key. If it finds an entry in the list matching k0, it returns the corresponding value. If it reaches the end of the linked list without finding a match, it returns null. The loop invariant for the while loop states the pertinent loop invariant property necessary to verify the postcondition of the method. Specifically, it indicates that the key k0 is bound to a value in contents if it is bound to the same value in the unsearched portion of the list. This loop invariant is sufficient to enable the automatic verification of the get() method.

Note that our implementation uses reference equality (=) instead of Java's .equals() method. This simplification allows us to avoid inter-class dependencies, and simplifies the verification. It is also possible to verify an implementation that uses .equals(), but the verified specification would need to refer to the meaning of .equals() as defined by the implementing class. A sample postcondition for an implementation of get that uses .equals() might be:

$$((\mathsf{result} \neq \mathsf{null}) \rightarrow (\exists \mathsf{k}.(\mathsf{k}, \mathsf{result}) \in \mathsf{contents} \wedge (\mathsf{equals\ k\ k0}))) \wedge$$
$$((\mathsf{result} = \mathsf{null}) \rightarrow \neg(\exists \mathsf{k\ v}.(\mathsf{k}, \mathsf{v}) \in \mathsf{contents} \wedge (\mathsf{equals\ k\ k0})))$$

where equals is a specification variable defining the meaning of .equals() from the class in which .equals() is specified. We use a similar technique in the specification of our hash table data structure to handle the use of the hashCode() method (see Chapter 7).

6.6.2 The remove() Method

Figure 6-5 presents the body of the remove() method. The remove() method takes a key k0, removes the corresponding mapping from the association list, and returns the previously bound value. It first invokes the private method _containsKey() on k0, to determine if the association list contains a mapping for k0. If so, it returns the result

```
1  private boolean _containsKey(Object k0)
2  /*: requires "theinvs"
3      ensures "result = (∃v.((k0, v) ∈ contents)) ∧ theinvs" */
4  {
5      Node current = first;
6      while /*: inv "(∃v.(k0, v) ∈ contents) =
7                     (∃v.(k0, v) ∈ current..con)" */
8          (current != null) {
9
10         if (current.key == k0)
11             return true;
12
13         current = current.next;
14     }
15     return false;
16 }
```

Figure 6-6: Body of AssociationList._containsKey()

of invoking the private method _remove(), which removes the mapping for k0 from the association list, and returns the previously bound value. Otherwise, remove() returns null.

6.6.3 The _containsKey() Method

Figure 6-6 presents the body of the _containsKey() method. The _containsKey() method takes a key k0 and returns true if k0 is a key in the association list, and false otherwise. The _containsKey() method uses a while loop to search through the nodes of the association list for k0. If it finds an entry in the list matching k0, it returns true. If it reaches the end of the linked list without finding a match, it returns false. The loop invariant for the while loop captures the pertinent loop invariant property necessary to verify the postcondition of the method. Specifically, it indicates that the key k0 is in contents if it is in the unsearched portion of the list. This loop invariant is sufficient to enable the automatic verification of the _containsKey() method.

6.6.4 The _remove() Method

Figure 6-7 presents the body of the _remove() method. The _remove() method takes a key k0 that is known to be in the association list and removes the corresponding key-value pair. It does this by first assigning a ghost variable v0 to the value to which k0 is bound. The purpose of this assignment is to name the value corresponding to the key k0; this name becomes useful later for removing the pair (k0, v0) from the nodes whose con component contains the pair.

The method then tests whether k0 is stored in the first node of the association list. If so, it removes that node by assigning the first field of the association list to the next node in the list. It also nulls out the next field of the removed node and updates

```
1  private Object _remove(Object k0)
2  /*: requires "k0 ≠ null ∧ (∃v. (k0, v) ∈ contents) ∧ theinvs"
3      modifies contents, first, next, con
4      ensures "contents = old contents - {(k0, result)} ∧
5               (k0, result) ∈ old contents ∧ theinvs" */
6  {
7      //: ghost specvar v0::obj;
8      //: havoc v0 suchThat "(k0, v0) ∈ contents";
9      Node f = first;
10     if (f.key == k0) {
11         Node second = f.next;
12         f.next = null;
13         //: "f..con" := "{(f..key, f..value)}";
14         first = second;
15         return f.value;
16     } else {
17         Node prev = first;
18         //: "prev..con" := "prev..con - {(k0, v0)}";
19         Node current = prev.next;
20         while /*: inv "prev ≠ null ∧
21                        prev..con = prev..(old con) - {(k0, v0)} ∧
22                        current ≠ null ∧ prev..next = current ∧
23                        prev ≠ current ∧
24                        contents = old contents - {(k0, v0)} ∧
25                        (∀n. n ∈ AssociationList ∧ n ∈ old alloc ∧
26                        n ≠ this → n..contents = old (n..contents)) ∧
27                        (k0, v0) ∈ current..con ∧
28                        comment ''ConDefInv''
29                        (∀n. n ∈ Node ∧ n ∈ alloc ∧ n ≠ null ∧ n ≠ prev →
30                        n..con = {(n..key, n..value)} ∪ n..next..con ∧
31                        (∀v. (n..key, v) ∉ n..next..con)) ∧
32                        (∀n. n..con = old (n..con) ∨
33                        n..con = old (n..con) - {(k0, v0)}) ∧
34                        null..con = ∅" */
35             (current.key != k0)
36             {
37                 //: "current..con" := "current..con - {(k0, v0)}";
38                 prev = current;
39                 current = current.next;
40             }
41         Node tmp = current.next;
42         prev.next = tmp;
43         current.next = null;
44         //: "current..con" := "{(current..key, current..value)}";
45         return current.value;
46     }
47 }
```

Figure 6-7: Body of AssociationList._remove()

the value of con for the removed node to restore the ConDef invariant. The method then returns the previously bound value.

If k0 is not stored in the first node, _remove() uses a while loop to search through the association list. At each iteration, it updates the con component of the currently examined node to remove the pair (k0, v0). It also keeps track of the previously examined node. When the method reaches the node that stores the mapping, it removes it from the list by assigning the next field of the previous node to the next field of the following node. It then nulls out the next field of the removed node and updates its con component to restore the ConDef invariant. The method then returns the previously bound value.

The loop invariant of the while loop captures the pertinent loop invariant property necessary to verify the postcondition of _remove(). Specifically, it keeps track of relevant properties about the previously examined and current nodes prev and curr, and properties about the abstract state component con, which is modified in the loop. To establish the ConDef invariant at the end of the method, the loop invariant captures: 1) a frame condition indicating that the ConDef invariant holds for all nodes other than prev (ConDefInv), 2) the value of con for prev, 3) that con for all objects is either unchanged, or else the result of removing the pair (k0, v0) from its original value, and 4) that the value of con for null is the empty set. The loop invariant additionally indicates that the value of contents is the result of removing (k0, v0) from its original value, for establishing the method postcondition. It also indicates that the value of contents is unchanged for all AssociationList objects other than the receiver, for verifying the frame condition for contents. Despite the complexity of this loop invariant, no proof commands are necessary to guide the provers in its verification. With this loop invariant, the provers are able to automatically verify the _remove() method.

6.6.5 Ease of Verification versus Efficiency

The association list implementation we have presented in this chapter is easy to understand and verify, but not the most efficient. The implementation of the remove() operation examines the list elements twice—first in the call to _containsKey(), to test if the key of interest is present, then, if it is, in the call to _remove(), to perform the actual removal. Although the asymptotic performance is the same within a constant factor, a more efficient implementation is clearly possible. (The put() method suffers from a similar inefficiency, and, for that reason, we provide the more efficient add() and replace() methods for contexts in which the client knows whether the given key is in the list.)

There are several reasons why we chose this design. First, by dividing the functionality of the remove() operation into the two parts implemented by _containsKey() and _remove(), we obtain smaller, well-encapsulated methods, which are in general easier to understand and verify. The _containsKey() and _remove() methods also implement functionality needed by methods other than remove(), resulting in better code reuse. By determining *a priori* whether there exists a binding for the given key, it also becomes possible to name the associated value, and to remove the corre-

sponding key-value pair incrementally from the con component of the earlier nodes in the list, as we do in the while loop of _remove(). In this approach, the ConDef invariant (which gives the contents of the linked list starting at a given node as the union of: 1) the singleton set containing the key-value pair at that node and 2) the contents of the next node) is always only violated at the currently examined node, and is incrementally restored with each iteration of the while loop. Without knowing whether a given key is bound in the association list, it would not be possible to update the con component of the earlier nodes in the list (since we have no way to refer to the value of the key-value pair to be removed) until the node containing the key is found. At that point, it would be necessary to update the con component of all the previous nodes in the list, violating the ConDef invariant for many different nodes simultaneously. To show that the ConDef invariant is restored after these updates, it would be necessary to keep track of all the nodes that need to be updated, and the aliasing relationships between them (or lack thereof). This, in turn, would most likely require the specification of additional abstract state, such as the set of nodes in use by the list, to successfully verify. In general, it is easier to verify incremental violations of a class invariant than to re-establish such an invariant for many objects simultaneously. Although our current design is not as efficient as this alternative approach, we chose it because it is conceptually simpler and easier to verify. Other association list implementations and specifications are also possible, including the use of shape analysis (instead of ghost variables) to specify reachability. We hope that the discussion of these two particular designs provide insight into some of the potential trade-offs between ease of verification and efficiency of implementation.

6.6.6 Summary

In this chapter, we show how we use Jahob to specify and verify an AssociationList class that implements a map. Our association list is implemented using a singly-linked list. The abstract state of the association list is expressed as a set of key-value pairs. In contrast to the priority queue example, which uses only dependent variables, the association list also uses ghost variables (whose values are explicitly updated using specification assignment statements) to specify its abstract state. We take this approach to demonstrate how to encode transitive closure as a recursive property, enabling the use of provers not able to handle the transitive closure operator.

While the implementation of the association list is straightforward, the invariants of the class capture non-trivial properties necessary for the correct implementation of the map interface. These invariants include injectivity invariants that express an acyclicity property and the lack of sharing between different lists. They also include invariants that guarantee that every key maps to a unique value. Despite this complexity, the association list verifies automatically, without the need for any proof commands. Because the verification conditions for the association list involve universal quantification, but no arithmetic properties, the first-order theorem prover SPASS is able to prove all the generated verification conditions without guidance.

The AssociationList class exports a number of different public operations, including the containsKey(), get(), put(), and remove() methods. These operations are supported

by a number of private methods, which are also verified. We show the complete implementation and specification for the get() and remove() methods, as well as all the methods on which they depend. The private _containsKey() and _remove() methods, in particular, include loop invariants that encode the necessary properties to ensure the correctness of the enclosing methods. The entire AssociationList class, as well as the supporting Node class, verifies automatically. We discuss the trade-off between efficiency of implementation and ease of verification that may occur in verifying data structure implementations, and describe an alternative implementation of association list that is more efficient but also potentially more difficult to verify.

Chapter 7

Hash Table

In this chapter, we describe a hash table data structure that we verified using Jahob. The hash table data structure, like the priority queue data structure, is one of the more difficult data structures to verify. It is both an array-based data structure, and a recursive data structure. It additionally maintains invariants that guarantee the correctness of the hash table operations, including injectivity invariants and invariants that ensure that the keys in the hash table are stored in the correct buckets. We use the hash table example to illustrate the specification and verification of a complex data structure that requires a substantial number of proof commands to verify.

Our Hashtable class implements a standard hash table data structure with separate chaining. The buckets of the hash table are implemented using singly-linked lists similar to the association list data structure from Chapter 6, but with a different specification that takes into account the properties of the enclosing hash table.

7.1 The Concrete State

Figure 7.1 presents part of the Hashtable class, as well as the Node class that stores the key-value pairs in the hash table. The concrete state of a hash table consists of an array table, where each entry contains either a linked list of Node objects or null. Each Node object consists of a key storing the key for a mapping in the hash table, a value storing the corresponding value, and a next field storing the next Node object in the linked list. The claimedby annotation in the declaration of the Node class indicates that only methods in the Hashtable class may modify the fields and specification variables of Node objects.

7.2 The Abstract State

The abstract state of a hash table consists of the ghost variables contents, the set of key-value pairs stored in the hash table, init, a boolean flag indicating whether the hash table is initialized, and con, the set of key-value pairs stored in the linked list starting at a given Node object. Unlike the association list specification, in which contents is a dependent specification variable, the contents specification variable in

```
public /*: claimedby Hashtable */ class Node {
    public Object key;
    public Object value;
    public Node next;
    /*: public ghost specvar con :: "(obj * obj) set" = "∅";
        invariant ConDef: "this ≠ null →
        con = {(key, value)} ∪ next..con ∧ (∀v. (key, v) ∉ next..con)";
        invariant ConNull: "null..con = ∅";
        invariant ConNonNull: "∀x y.(x, y) ∈ con → x ≠ null ∧ y ≠ null";
        invariant ConAlloc: "∀x y.(x,y) ∈ con → x ∈ alloc ∧ y ∈ alloc";
    */
}
public class Hashtable {
    private Node[] table = null;
    /*:
      public ghost specvar contents :: "(obj * obj) set" = "∅";
      public ghost specvar init :: "bool" = "False";

      static specvar h :: "(obj ⇒ int ⇒ int)";
      vardefs "h == (λo1. (λi1. ((abs (hashFunc o1)) mod i1)))";
      static specvar abs :: "(int ⇒ int)"
      vardefs "abs == (λi1. (if (i1 < 0) then (−i1) else i1))";

      invariant ContentsDefInv: "init →
      contents = {(k,v). (k,v) ∈ table.[(h k (table..length))]..con}";
      invariant Coherence: "init → (∀i k v. 0 ≤ i ∧ i < table..length →
      (k,v) ∈ table.[i]..con → h k (table..length) = i)";
      invariant MapInv:
      "∀k v0 v1. (k,v0) ∈ contents ∧ (k,v1) ∈ contents → v0 = v1";
      invariant TableNotNull: "init → table ≠ null";
      invariant TableHidden: "init → table ∈ hidden";
      invariant NodeHidden1: "init → (∀i. 0 ≤ i ∧ i < table..length ∧
      table.[i] ≠ null → table.[i] ∈ hidden)";
      invariant NodeHidden2: "∀n. n ∈ Node ∧ n ∈ alloc ∧ n ≠ null ∧
      n..next ≠ null → n..next ∈ hidden";
      invariant HashInv: "init → (∀k. 0 ≤ (h k (table..length)) ∧
      (h k (table..length)) < table..length)";
      invariant FirstInjInv: "init → (∀i x y. y = x..next ∧ y ≠ null ∧
      0 ≤ i ∧ i < table..length → y ≠ table.[i])";
      invariant NextInjInv:
      "∀x1 x2 y. y ≠ null ∧ y = x1..next ∧ y = x2..next → x1 = x2";
      invariant ElementInjInv: "init → (∀ht i j. ht ∈ Hashtable ∧
      ht ∈ alloc ∧ ht..init ∧ 0 ≤ i ∧ i < ht..table..length ∧ 0 ≤ j ∧
      j < table..length ∧ ht..table.[i] = table.[j] ∧
      table.[j] ≠ null → ht = this ∧ i = j)";
      invariant TableInjInv:
      "∀ht. ht..table = table ∧ table ≠ null → ht = this"; */
    ...
}
```

Figure 7-1: Hashtable Example

this case is a ghost variable that is updated using specification assignment statements. The initial values of contents, init, and con are all set to ensure that the hash table invariants hold in the initial program state.

The specification also declares the static specification variables h and abs as shorthands that denote the hash and absolute value functions, respectively. The hash function h is a lambda expression that takes an object o1 to be hashed, and an integer value i1 and produces an integer in the range $[0, i1)$ based on the hash code of o1. The absolute value function abs is a lambda expression that takes an integer and returns its absolute value.

7.3 Abstraction Function and Invariants

The abstract state of a hash table consists of ghost variables, with the static dependent variables h and abs defining shorthands used in other parts of the specification. The vardefs declarations give the definition for these shorthands. The vardefs declaration for h defines it as a lambda expression that takes an object and an integer bound, and produces the mod of the absolute value of that hash code with respect to the given bound. (The variable hashFunc on which h depends is a specification variable separately declared in the Object class, which represents the hash function for the given object.) The vardefs declarations for abs define it as the absolute value function with the standard meaning.

Since the abstract state of a hash table consists of ghost variables, the abstraction function is expressed in terms of data structure invariants. The ConDef and ConNull invariants give the definition for the abstract state component con. The ConDef invariant indicates that, for an allocated node, con consists of the union of the singleton set containing the key-value pair stored at that node, and the value of con for the next node in the linked list. It additionally indicates that next..con does not contain a value for the key stored at the current node, thus ensuring that a key occurs at most once in a bucket. The ConNull invariant indicates that null..con is the empty set. This is similar to the corresponding invariants of the same name in the association list example, but expressed within the Node class, so that instead of writing ConDef as being explicitly quantified over allocated Node objects, it is automatically quantified by Jahob based on the implicit references to the receiver. The resulting invariants are the same.

In the Hashtable class, the ContentsDefInv invariant gives the definition for contents in terms of the abstract state component con. It indicates that, for an initialized hash table, contents consists of the key-value pairs in cons for the buckets in the table, considering only those pairs that are hashed to the correct bucket. The Coherence invariant additionally requires that every key-value pair in the hash table be in the correct bucket. The MapInv invariant indicates that the hash table contains no more than one mapping for a given key.

In addition to the invariants that comprise the abstraction function, the Hashtable class contains the following invariants which specify the valid concrete and abstract states of the hash table data structure:

- The ConNonNull invariant indicates that the key and value in every key-value pair in con are non-null.
- The ConAlloc invariant indicates that the key and value in every key-value pair in con are allocated objects.
- The TableNonNull invariant indicates that, for an initialized hash table, table is non-null.
- The TableHidden invariant indicates that, for an initialized hash table, table is hidden.
- The NodeHidden1 invariant indicates that, for an initialized hash table, the first Node object in every bucket is hidden. The NodeHidden2 invariant indicates that if an allocated Node object is hidden, and its next field refers to a Node object, then that Node object is also hidden. Together, these two invariants guarantee that all Node objects used by the hash table are hidden.
- The HashInv invariant indicates that, for an initialized hash table, the result of the hash function h is a value that is within the array bounds of table (i.e. in the range [0, table..length)).
- The FirstInjInv invariant indicates that, for an initialized hash table, the first Node object in a bucket is not pointed to by the next field of any Node object. The NextInjInv invariant indicates that next is injective—a Node object is pointed to by the next field of at most one Node object. Together, these two invariants guarantee that every non-empty bucket in the hash table contains an acyclic linked list.
- The ElementInjInv invariant indicates that, for an initialized hash table, the first Node object in a bucket is not the first Node object in any other bucket in either this hash table or any other hash table.
- The TableInjInv invariant indicates that, for an initialized hash table, that table is not shared by any other hash table. Together, the FirstInjInv, NextInjInv, ElementInjInv, and TableInjInv invariants guarantee that the table array and Node objects belonging to a hash table are not shared by any other hash table.

For invariants in the Hashtable class that contain free implicit or explicit references to the receiver this, Jahob automatically quantifies the invariant over all allocated Hashtable objects. These invariants apply to all allocated instances of Hashtable. As with the priority queue and association list invariants, the hash table invariants capture injectivity properties that express the lack of aliasing, properties that ensure the lack of null dereferences and array bounds exceptions, and encapsulation properties involving the class' hidden set.

Note that the invariants ContentsDefInv, Coherence, and HashInv depend on the specification variable h, which is defined in terms of the specification variable hashFunc from the Object class. This inter-class dependency means that changes to the

state of the Object class may result in violations of the hash table invariants. In our case, hashFunc is defined as the value of a final hashCode field for objects, so that this problem does not occur. But in general, inter-class dependencies may result in class invariants that need to be re-asserted at method boundaries of external classes to ensure soundness. For example, if the hash code of an object in the hash table were to change, and the relevant invariants not re-established, then subsequent searches of the hash table may not find the object, resulting in incorrect behavior. Appropriately re-asserting invariants would identify this problem, as would specifying a policy disallowing the modification of objects in the hash table, which can be done using history variables.

7.4 Method Contracts for Public Methods

Figure 7-2 presents the method contracts for all the public methods of the Hashtable class. These methods constitute the interface that the Hashtable class exports to its clients. The interfaces for the Hashtable constructor and the containsKey(), get(), isEmpty(), put(), and remove() methods are modeled after the interfaces for the corresponding methods in java.util.Hashtable. The methods add() and replace() are methods not found in java.util.Hashtable. We include these methods because they offer alternative interfaces that clients may find useful.

7.4.1 The Hashtable Constructor

The Hashtable constructor is a public constructor that creates a new, empty, initialized Hashtable object. The modifies clause indicates that it may modify the init and contents components of the hash table under construction. The ensures clause indicates that the resulting hash table is initialized and empty. The lack of a requires clause indicates that the constructor has no precondition.

7.4.2 The containsKey() Method

The containsKey() method is a public instance method that takes a key k0 and returns a boolean value indicating whether k0 is a key in the hash table. The requires clause indicates that containsKey() must be invoked on an initialized hash table, and that k0 must be non-null. The ensures clause indicates that the method returns true if k0 corresponds to a mapping in the hash table, and false otherwise. There is no modifies clause because containsKey() does not modify the program state.

7.4.3 The get() Method

The get() method is a public instance method that takes a key k0 and returns the value to which k0 is mapped in the hash table. The requires clause indicates that get() must be invoked on an initialized hash table, and that k0 must be non-null. The ensures clause indicates that if get() returns null, then the hash table does not

```
public Hashtable()
/*: modifies contents, init
    ensures "init ∧ contents = ∅" */
{ ... }

public boolean containsKey(Object k0)
/*: requires "init ∧ k0 ≠ null"
    ensures "result = (∃v.((k0, v) ∈ contents))" */
{ ... }

public Object get(Object k0)
/*: requires "init ∧ k0 ≠ null"
    ensures "(result ≠ null → (k0, result) ∈ contents) ∧
             (result = null → ¬(∃v.(k0, v) ∈ contents))" */
{ ... }

public boolean isEmpty()
/*: requires "init"
    ensures "result = (contents = ∅)" */
{ ... }

public Object put(Object k0, Object v0)
/*: requires "init ∧ k0 ≠ null ∧ v0 ≠ null"
    modifies contents
    ensures "contents = old contents - {(k0, result)} ∪ {(k0, v0)} ∧
             (result = null → ¬(∃v.(k0, v) ∈ old contents)) ∧
             (result ≠ null → (k0, result) ∈ old contents)" */
{ ... }

public Object remove(Object k0)
/*: requires "init ∧ k0 ≠ null"
    modifies contents
    ensures "contents = old contents - {(k0, result)} ∧
             (result = null → ¬(∃v.(k0,v) ∈ old contents)) ∧
             (result ≠ null → (k0,result) ∈ old contents)" */
{ ... }

public void add(Object k0, Object v0)
/*: requires "init ∧ k0 ≠ null ∧ v0 ≠ null ∧ ¬(∃v.(k0, v) ∈ contents)"
    modifies contents
    ensures "contents = old contents ∪ {(k0, v0)}" */
{ ... }

public Object replace(Object k0, Object v0)
/*: requires "init ∧ k0 ≠ null ∧ v0 ≠ null ∧ (∃v.(k0, v) ∈ contents)"
    modifies contents
    ensures "contents = old contents - {(k0, result)} ∪ {(k0, v0)} ∧
             (k0, result) ∈ old contents" */
{ ... }
```

Figure 7-2: Exported Operations of the Hashtable Class

contain a mapping for k0. But if get() returns a non-null value, then the return value is the value to which k0 is mapped in the hash table. There is no modifies clause because get() does not modify the program state.

7.4.4 The isEmpty() Method

The isEmpty() method is a public instance method that returns a boolean value indicating whether the given hash table is empty. The requires clause indicates that isEmpty() must be invoked on an initialized hash table. The ensures clause indicates that the method returns true if the hash table is empty, and false otherwise. There is no modifies clause because isEmpty() does not modify the program state.

7.4.5 The put() Method

The put() method is a public instance method that adds a key-value pair to the hash table. The requires clause for put() indicates that put() must be invoked on an initialized hash table, and that the key k0 and value v0 being added must be non-null. The modifies clause indicates that put() may modify the contents of the given hash table. The ensures clause indicates that the value of contents after invoking put() is the result of removing the pair (k0, result) from contents, and adding the pair (k0, v0). It also indicates that if the return value is null, then the hash table did not originally contain a mapping for the key k0. But if the return value is not null, then it is the value to which k0 was originally mapped.

7.4.6 The remove() Method

The remove() method is a public instance method that removes a key-value pair from the hash table. The requires clause indicates that remove() must be invoked on an initialized hash table, and that the key k0 being removed must be non-null. The modifies clause indicates that remove() may modify the contents of the given hash table. The ensures clause indicates that the value of contents after invoking remove() is the result of removing the pair (k0, result) from contents. As is the case for put(), the ensures clause also indicates that if the return value is null, then the hash table did not originally contain a mapping for k0. But if the return value is not null, then it is the value to which k0 was originally mapped.

7.4.7 The add() Method

The add() method is a public instance method that adds a key-value pair to the hash table. It is similar to the put() method, but requires that the hash table contain no previous binding for the given key. It is more efficient than put() and offers an alternative interface for adding a binding to the hash table when its precondition is known to hold. Its requires clause indicates that add() must be invoked on an initialized Hashtable, that the key k0 and value v0 must be non-null, and that the key k0 is not already bound to a value in the hash table. The modifies clause indicates

```
private int compute_hash(Object o1)
/*: requires "init ∧ o1 ≠ null ∧ theinvs"
    ensures "result = h o1 (table..length) ∧ 0 ≤ result ∧
             result < table..length ∧ alloc = old alloc ∧ theinvs" */
{ ... }

private boolean _containsKey(Object k0)
/*: requires "init ∧ k0 ≠ null ∧ theinvs"
    ensures "result = (∃v.((k0, v) ∈ contents)) ∧ theinvs" */
{ ... }

private boolean bucketContainsKey(int bucket_id, Object k0)
/*: requires "init ∧ 0 ≤ bucket_id ∧ bucket_id < table..length ∧
              theinvs"
    ensures "result = (∃v.((k0, v) ∈ table.[bucket_id]..con)) ∧
             theinvs" */
{ ... }

private Object _remove(Object k0)
/*: requires "(comment ''Init'' init) ∧ k0 ≠ null ∧
              (∃v.(k0, v) ∈ contents) ∧ theinvs"
    modifies contents, con, next, arrayState
    ensures "(contents = old contents - {(k0, result)}) ∧
             ((k0, result) ∈ old contents) ∧
             (∀a i.a ≠ table → a.[i] = old (a.[i])) ∧ theinvs" */
{ ... }
```

Figure 7-3: Private Methods of the Hashtable Class (continued in Figure 7-4)

that it may modify the contents of the given hash table. The ensures clause indicates that the method adds the given key-value pair to the hash table contents.

7.4.8 The replace() Method

The replace() method is a public instance method that binds an existing key in the hash table to a new value. It is similar to the put() method, but requires that the hash table contain a previous binding for the given key. It is more efficient than put() and offers an alternative interface for adding a binding to the hash table when its precondition is known to hold. Its requires clause indicates that replace() must be invoked on an initialized Hashtable, that the key k0 and value v0 being added must be non-null, and that k0 must be already bound to a value in the hash table. The modifies clause indicates that replace() may modify the contents of the given hash table. The ensures clause indicates that the replace() removes the previous binding for k0 from contents, adds the given key-value pair, and returns the previously bound value.

```
private Object removeFirst(Object k0, int hc)
/*: requires "init ∧ k0 ≠ null ∧ (∃v.(k0, v) ∈ contents) ∧
             comment ''KFound'' (k0 = table.[hc]..key) ∧
             comment ''HCProps'' (0 ≤ hc ∧ hc < table..length ∧
             hc = h k0 (table..length)) ∧ theinvs"
    modifies contents, con, next, arrayState
    ensures "(contents = old contents - {(k0, result)}) ∧
            ((k0, result) ∈ old contents) ∧
            (∀a i.a ≠ table → a.[i] = old (a.[i])) ∧ theinvs" */
{ ... }

private Object removeFromBucket(Object k0, int hc)
/*: requires "comment ''Init'' init ∧ k0 ≠ null ∧
             (∃v.(k0, v) ∈ contents) ∧
             comment ''KNotFound'' (k0 ≠ table.[hc]..key) ∧
             comment ''HCProps'' (0 ≤ hc ∧ hc < table..length ∧
             hc = h k0 (table..length)) ∧ theinvs"
    modifies contents, con, next, arrayState
    ensures "(contents = old contents - {(k0, result)}) ∧
            ((k0, result) ∈ old contents) ∧
            (∀a i. a ≠ table → a.[i] = old (a.[i])) ∧ theinvs" */
{ ... }

private void _add(Object k0, Object v0)
/*: requires "comment ''Init'' init ∧ k0 ≠ null ∧ v0 ≠ null ∧
             ¬(∃v.(k0, v) ∈ contents) ∧ theinvs"
    modifies contents, arrayState, "new..con", "new..next",
            "new..value", "new..key"
    ensures  "contents = old contents ∪ {(k0, v0)} ∧
             (∀a i. a ≠ table → a.[i] = old (a.[i])) ∧ theinvs" */
{ ... }
```

Figure 7-4: Private Methods of the Hashtable Class (continued from Figure 7-3)

7.5 Method Contracts for Private Methods

Figures 7-3 and 7-4 presents the method contracts for the private methods of the Hashtable class. These methods implement various functionalities that are needed to support the public operations of the Hashtable class. Our experience indicates that methods that modify state are substantially more difficult to verify than methods that either do not modify state, or do so indirectly, by invoking other methods. For this reason, the code in the Hashtable class that modify state is divided amongst many small methods with well-defined interfaces. These methods are then invoked by either private or public methods higher up in the call graph to obtain the desired higher level functionality of the hash table.

7.5.1 The compute_hash() Method

The compute_hash() method is a private instance method that takes an object o1 and hashes it to produce a valid index into table. It is used by all the methods in the Hashtable class that need to compute the index corresponding to the bucket for a given key. The requires clause indicates that compute_hash() must be invoked on an initialized hash table, that o1 must be non-null, and that the invariants of the hash table must hold. The ensures clause indicates that the return value is the effect of applying the hash function h on o1 and table..length and that this value is within the range $[0, \text{table..length})$. It also indicates that compute_hash() does not allocate any objects, and that it preserves the invariants of the hash table. There is no modifies clause because compute_hash() does not modify the program state.

7.5.2 The _containsKey() Method

The _containsKey() method is a private instance method that takes a key k0 and returns a boolean value indicating whether k0 is a key in the hash table. It implements the functionality needed by the public containsKey() method, and is also invoked by the put() and remove() methods. The requires clause indicates that _containsKey() must be invoked on an initialized hash table, that k0 must be non-null, and that the invariants of the hash table must hold. The ensures clause indicates that _containsKey() returns true if k0 corresponds to a key-value pair in the hash table, and false otherwise. It also indicates that _containsKey() preserves the hash table invariants. There is no modifies clause because _containsKey() does not modify the program state.

7.5.3 The bucketContainsKey() Method

The bucketContainsKey() method is a private instance method that takes an index bucket_id and a key k0 and returns a boolean value indicating whether k0 maps to a value in the bucket at bucket_id. It is a helper method that implements part of the functionality of the _containsKey() method. The requires clause indicates that bucketContainsKey() must be invoked on an initialized hash table, that bucket_id must be within the range $[0, \text{table..length}]$, and that the invariants of the hash table must hold.

The ensures clause indicates that the method returns true if k0 maps to a value in the bucket at bucket_id, and false otherwise. It also indicates that the method preserves invariants of the hash table. There is no modifies clause because bucketContainsKey() does not modify the program state.

7.5.4 The _remove() Method

The _remove() method is a private instance method that takes a key k0 that is known to be in the hash table and removes the corresponding mapping from the hash table. It is a helper method that is invoked by the remove(), put(), and replace() methods. The requires clause indicates that _remove() must be invoked on an initialized hash table, that k0 must be non-null and have a corresponding value in contents, and that the invariants of the hash table must hold. The clause that specifies that the hash table must be initialized has the label Init, which is used in the body of the method for its verification. The modifies clause indicates that the _remove() method may modify the contents of the hash table, the Node.con and Node.next fields, and the contents of arrays. The ensures clause indicates that the effect of invoking _remove() is to remove the pair (k0, result) from contents and that this pair was originally in contents. It also indicates that, with the exception of this.table, all other arrays are unchanged, and that _remove() re-establishes the invariants of the hash table class at the end of the method.

7.5.5 The removeFirst() Method

The removeFirst() method is a private instance method that removes a key k0 and its corresponding value from the hash table given that the mapping for k0 is stored in the first node in the bucket at index hc. It is a helper method that implements part of the functionality of _remove(). The requires clause indicates that removeFirst() must be invoked on an initialized hash table, that k0 must be non-null, and that the hash table must contain a mapping for k0. It also indicates that the mapping for k0 is stored in the first node of its bucket at index hc, that hc is within the bounds [0, table..length) and is the result of applying the hash function h to k0 and table..length, and that the hash table invariants must hold. The modifies clause indicates that removeFirst() may modify the contents of the hash table, the Node.con and Node.next fields, and the contents of arrays. The ensures clause indicates that the effect of invoking removeFirst() is to remove the pair (k0, result) from contents and that this pair was originally in contents. It also indicates that with the exception of table, all other arrays are unchanged, and that removeFirst() re-establishes the invariants of the hash table class at the end of the method.

As in the labeled clause (Init) in the precondition of _remove(), the comment operator in the precondition applies the given labels to the conjuncts of the precondition. Proof commands in the body of the method can then refer to individual conjuncts in their from clause to enable the verification of the method.

7.5.6 The removeFromBucket() Method

The removeFromBucket() method is a private instance method that removes a key k0 and its corresponding value from the hash table given that the mapping for k0 is not stored in the first node of its bucket. As with removeFirst(), it is also a helper method that implements part of the functionality of _remove(). The requires clause indicates that removeFromBucket must be invoked on an initialized hash table, and that k0 must be non-null. It also indicates that the hash table must contain a mapping for k0, that this mapping is not stored in the first node of the bucket at index hc, that hc is within the bounds [0, table..length) and is the result of applying the hash function h to k0 and table..length, and that the hash table invariants must hold. The modifies clause indicates that removeFromBucket() may modify the contents of the hash table, the Node.con and Node.next fields, and the contents of arrays. The ensures clause indicates that the effect of invoking removeFromBucket() is to remove the pair (k0, result) from contents and that this pair was originally in contents. It also indicates that with the exception of this.table, all other arrays are unchanged, and that removeFromBucket() re-establishes the invariants of the hash table at the end of the method.

7.5.7 The _add() Method

The _add() method is a private instance method that adds a binding to the hash table for a key k0 for which the hash table does not already contain a binding. It implements the functionality of the add() method and is also invoked by put() and replace(). The requires clause of _add() indicates that it must be invoked on an initialized hash table, that the key k0 and value v0 to be added must be non-null, that the hash table must not already contain a binding for k0, and that the invariants of the hash table hold. The modifies clause indicates that _add() may modify the contents of the hash table, the contents of arrays, as well as the con, next, value, and key components of objects that it allocates. The ensures clause indicates that the effect of invoking _add() is to add the pair (k0, v0) to the contents of the hash table. It also indicates that with the exception of this.table, all other arrays are unchanged, and that _add() re-establishes the invariants of the hash table at the end of the method.

7.6 Implementation and Verification

Of the data structures we verified using Jahob, the hash table data structure was probably one of the most difficult. While observer methods and methods that did not modify state directly contain little or no lines of proof, many methods that modify state contain more lines of proof than code. This section illustrates the implementation and verification of the Hashtable class using the example of the get() and remove() methods. It describes the implementation and verification of these methods and of all the methods in the Hashtable class that they invoke either directly or transitively.

```
1  public Object get(Object k0)
2  /*: requires "init ∧ k0 ≠ null"
3      ensures "(result ≠ null → (k0, result) ∈ contents) ∧
4               (result = null → ¬(∃ v. (k0, v) ∈ contents))" */
5  {
6      //: instantiate "theinv ContentsDefInv" with "this";
7      /*: mp ThisContentsDef:
8          "this ∈ alloc ∧ this ∈ Hashtable ∧ init → contents =
9           {(k, v). (k, v) ∈ table.[(h k (table..length))]..con}"; */
10
11     int hc = compute_hash(k0);
12     Node curr = table[hc];
13
14     //: note HCDef: "hc = h k0 (table..length)";
15     /*: note InCurr: "∀v.(((k0, v) ∈ contents) = ((k0, v) ∈ curr..con))"
16         from ThisContentsDef, HCDef; */
17
18     while /*: inv "∀v.((k0, v) ∈ contents) = ((k0, v) ∈ curr..con)" */
19         (curr != null) {
20
21         if (curr.key == k0)
22             return curr.value;
23
24         curr = curr.next;
25     }
26     return null;
27 }
```

Figure 7-5: Body of Hashtable.get()

7.6.1 The get() Method

Figure 7-5 presents the body of the get() method. The get() method starts by invoking the compute_hash() method on the key k0 to obtain an index hc. This index corresponds to the bucket that would contain k0, if k0 were in the hash table. It then uses a while loop to search through the linked list at that bucket for k0. If it finds an entry in the linked list matching k0, it returns the corresponding value. If it reaches the end of the linked list without finding a match, it returns null.

Verifying the Loop Invariant

The body of get() contains several proof commands that provide the guidance that the provers need to verify the method's loop invariant. The loop invariant states the following property: there is a value corresponding to the key k0 in contents if and only if such a value is in the unsearched portion of the linked list. Without proof commands, Jahob is unable to verify that the loop invariant holds in the base case—when the method first reaches the top of the loop.

The proof of this property is based on the definition of contents as given by the ContentsDefInv invariant, which states that all keys are hashed to the correct

```
1  private int compute_hash(Object o1)
2  /*: requires "init ∧ o1 ≠ null ∧ theinvs"
3      ensures "result = h o1 (table..length) ∧ 0 ≤ result ∧
4               result < table..length ∧ alloc = old alloc ∧ theinvs" */
5  {
6      int hc = o1.hashCode();
7
8      if (hc < 0) { hc = -hc; }
9
10     //: note LengthPos: "0 < table..length";
11     /*: note ResLt: "(hc mod table..length) < table..length"
12         from TrueBranch, FalseBranch, LengthPos; */
13
14     return (hc % table.length);
15 }
```

Figure 7-6: Body of Hashtable.compute_hash()

```
public int hashCode()
//: ensures "result = hashFunc this ∧ alloc = old alloc"
{ ... }
```

Figure 7-7: Method contract for Object.hashCode()

bucket. The instantiate command at line 6 therefore directs Jahob to instantiate the ContentsDefInv invariant (which is universally quantified) with the receiver this to obtain the proved lemma this $\in$ alloc $\wedge$ this $\in$ Hashtable $\wedge$ init $\rightarrow$ contents $=$ (k, v).(k, v) $\in$ table.[(h k (table..length))]..con. The mp command at line 7 then directs Jahob to apply *modus ponens* to this result to establish the right-hand side of the implication and to label it ThisContentsDef. To apply this result in the proof, it is necessary to show that the computed index hc corresponds to the correct bucket for k0. The note command at line 14 therefore directs Jahob to show that the index hc is equal to h k0 (table..length), and to label the proved lemma HCDef. Finally, the note command at line 15 directs Jahob to use the proved lemmas ThisContentsDef and HCDef to conclude that the loop invariant holds.

7.6.2 The compute_hash() Method

Figure 7-6 presents the body of the compute_hash() method, which is invoked by several methods of the Hashtable class, including get(). The compute_hash() method takes an object o1 and returns an index into table corresponding to the bucket for o1. It invokes Object.hashCode() on o1, takes the absolute value of the result, then returns the mod of the absolute value with respect to table.length. To generate the proof obligations for compute_hash(), Jahob requires the method contract for the invoked hashCode() method from the Object class (see Figure 7-7). The hashCode() method

```
1  public Object remove(Object k0)
2  /*: requires "init ∧ k0 ≠ null"
3      modifies contents
4      ensures "contents = old contents - {(k0, result)} ∧
5               (result = null → ¬(∃v0. (k0,v0) ∈ old contents)) ∧
6               (result ≠ null → (k0,result) ∈ old contents)" */
7  {
8      if (!_containsKey(k0))
9          return null;
10     else
11         return _remove(k0);
12 }
```

Figure 7-8: Body of Hashtable.remove()

returns the result of applying the object hash function to the receiver. It does not allocate any objects, nor does it modify any publicly visible state.

Verifying the Postcondition

While most of the proof obligations for the compute_hash() method prove without proof commands, one of the conjuncts in the postcondition does not. This conjunct addresses the property that the return value is less than the length of table. The body of compute_hash() contains two note commands that provide the guidance that the provers need to establish this conjunct.

First, the note command at line 10 directs Jahob to establish that table..length is greater than zero, and to label the proved lemma LengthPos. The note command at line 11 then directs Jahob to use the labeled facts TrueBranch, FalseBranch and LengthPos to prove that the return value is less than table..length. TrueBranch refers to the branch condition hc < 0, while FalseBranch refers to its negation $\neg$(hc < 0). Depending on which path Jahob is verifying, only one of these two conditions will hold. Jahob automatically applies the condition appropriate to the path and omits the one that does not hold. For both branches, these two note commands are sufficient to guide the provers to a successful proof of the desired property.

7.6.3 The remove() Method

Figure 7-8 presents the body of the remove() method. The remove() method takes a key k0, removes the corresponding key-value pair from the hash table, and returns the previously bound value. It first tests whether k0 is a key in the hash table by invoking the private method _containsKey(). If k0 is not in the hash table, remove() returns null. Otherwise, it returns the result of invoking the private method _remove(). The _remove() method performs the necessary removal, and returns the previously bound value.

The provers are able to discharge all the proof obligations for the remove() method without any guidance from proof commands. In general, we have found that Jahob is

```
1  private boolean _containsKey(Object k0)
2  /*: requires "init ∧ k0 ≠ null ∧ theinvs"
3      ensures "result = (∃v.((k0, v) ∈ contents)) ∧ theinvs" */
4  {
5      //: instantiate "theinv ContentsDefInv" with "this";
6      /*: mp ThisContentsDef:
7          "this ∈ alloc ∧ this ∈ Hashtable ∧ init → contents =
8           {(k,v). (k, v) ∈ table.[(h k (table..length))]..con}"; */
9
10     int hc = compute_hash(k0);
11     boolean res = bucketContainsKey(hc, k0);
12
13     //: note HCDef: "hc = h k0 (table..length)";
14     //: note InCon: "res = (∃v.((k0, v) ∈ table.[hc]..con))";
15     /*: note ShowResult: "res = (∃v.((k0, v) ∈ contents))"
16         from InCon, HCDef, ThisContentsDef; */
17
18     return res;
19 }
```

Figure 7-9: Body of Hashtable._containsKey()

often able to verify methods like remove()—methods that only modify state indirectly (through called methods)—without proof commands.

7.6.4 The _containsKey() Method

Figure 7-9 presents the body of the _containsKey() method. The _containsKey() method takes a key k0 and returns true if k0 is a key in the hash table, and false otherwise. It first invokes compute_hash() on k0 to obtain an index hc into the appropriate bucket in the hash table. It then returns the result of invoking bucketContainsKey() on hc and k0.

Verifying the Postcondition

The body of _containsKey() contains several proof commands that provide the guidance that the provers need to verify the method postcondition. Specifically, the proof commands address the property that the return value is true if k0 has a corresponding value in the hash table, and false otherwise. This property depends primarily on the invariant ContentsDefInv. Its proof is similar to the proof of the loop invariant in get().

The instantiate command at line 5 first directs Jahob to instantiate ContentsDefInv with the receiver object, obtaining the proved lemma this ∈ alloc ∧ this ∈ Hashtable ∧ init → contents = (k, v).(k, v) ∈ table.[(h k (table..length))]..con. The mp command at line 6 then directs Jahob to apply *modus ponens* to the result to obtain the consequent of the implication—that the contents of the receiver object is equal to {(k, v).(k, v) ∈ table.[(h k (table..length))]..con}—and to label the proved lemma ThisContentsDef. To

```
1  private boolean bucketContainsKey(int bucket_id, Object k0)
2  /*: requires "init ∧ 0 ≤ bucket_id ∧ bucket_id < table..length ∧
3                theinvs"
4      ensures "result = (∃v.((k0, v) ∈ table.[bucket_id]..con)) ∧
5               theinvs" */
6  {
7      Node curr = table[bucket_id];
8      while /*: inv "(∃v.(k0, v) ∈ table.[bucket_id]..con) =
9                     (∃v.(k0, v) ∈ curr..con)" */
10         (curr != null) {
11
12         if (curr.key == k0)
13             return true;
14
15         curr = curr.next;
16     }
17     return false;
18 }
```

Figure 7-10: Body of Hashtable.bucketContainsKey()

apply this lemma, it is necessary to show that hc corresponds to the result of applying the hash function h to k0 and table..length. The note command at line 13 therefore directs Jahob to prove this property, and to label the proved lemma HCDef. The note command at line 14 then directs Jahob to prove an intermediate lemma—that the return value of bucketContainsKey() is true if k0 is a key in the bucket at hc, and false otherwise—and to label the proved lemma InCon. Finally, the note command at line 15 directs Jahob to use the proved lemmas InCon, HCDef, and ThisContentsDef to prove that the value returned by bucketContainsKey() corresponds to whether k0 is in contents. Together, these proof commands provide the necessary guidance for Jahob to verify the postcondition of _containsKey().

7.6.5 The bucketContainsKey() Method

Figure 7-10 presents the body of the bucketContainsKey() method, which implements part of the functionality of _containsKey(). The bucketContainsKey() method takes the index of a bucket in the hash table and a key k0, and returns a boolean value indicating whether k0 corresponds to a key-value pair in the bucket. It uses a while loop to search through the linked list at the given bucket, starting with the first node. If k0 is equal to the key at the node currently under consideration, then the method returns true. If the method reaches the end of the linked list without finding k0, then it returns false. The loop invariant for the while loop consists of the property that there is a key-value pair corresponding to the key k0 in the given bucket if and only if such a pair is in the unsearched portion of the linked list. No proof commands are necessary for Jahob to verify this method, which is not surprising for a method that does not modify the program state.

```
1  private Object _remove(Object k0)
2  /*: requires "(comment ''Init'' init) ∧ k0 ≠ null ∧
3                 (∃v.(k0, v) ∈ contents) ∧ theinvs"
4      modifies contents, con, next, arrayState
5      ensures "(contents = old contents - {(k0, result)}) ∧
6               ((k0, result) ∈ old contents) ∧
7               (∀a i.a ≠ table → a.[i] = old (a.[i])) ∧ theinvs" */
8  {
9      //: ghost specvar v0::obj;
10     //: havoc v0 suchThat KeyInContents: "(k0, v0) ∈ contents";
11
12     int hc = compute_hash(k0);
13     Node f = table[hc];
14
15     //: note ThisProps: "this ∈ alloc ∧ this ∈ Hashtable"; */
16     //: note HCProps: "hc = h k0 (table..length)";
17     /*: note KeyInBucket: "(k0, v0) ∈ table.[hc]..con"
18         from KeyInContents, ContentsDef, Init, ThisProps, HCDef; */
19
20     if (f.key == k0)
21         return removeFirst(k0, hc);
22     else
23         return removeFromBucket(k0, hc);
24 }
```

Figure 7-11: Body of Hashtable._remove()

7.6.6 The _remove() Method

Figure 7-11 presents the body of the _remove() method. The _remove() method takes a key k0 that is known to be in the hash table and removes the corresponding key-value pair. It does this by first invoking the compute_hash() method on k0 to obtain the index to the appropriate bucket. It then tests whether the first node in the linked list corresponds to the entry for k0. If so, it returns the result of the removeFirst() method, which removes the first node in the bucket, and returns the corresponding value. Otherwise, it returns the result of removeFromBucket(), which finds and removes the node for k0 from the bucket, and returns the corresponding value.

Verifying that f is Non-Null

The body of _remove() contains several proof commands that provide the guidance that the provers need to verify _remove(). Specifically, the proof commands address the null pointer check for the dereference of f in line 20, in the test for the if statement. The local variable f refers to the reference at index hc of table. If f were null, then the bucket at that index would be empty. However, k0 hashes to that bucket, and is known to be in the hash table, so f must be non-null. The proof commands show that the key-value pair corresponding to k0 is indeed in that bucket.

To simplify the proof, the _remove() method first declares a ghost variable v0. The havoc command at line 10 then assigns v0 to the value corresponding to k0

in contents, and labels this property KeyInContents. The precondition of _remove() guarantees that such a value exists; naming it eliminates the need for using existential quantifiers in the proof, which in turn simplifies the proof task for the provers.

The proof depends primarily on the ContentsDefInv invariant. One way to apply ContentsDefInv to the receiver object is to use an instantiate command followed by an mp command, as is done in get() and _containsKey(). This method uses an alternative but equally effective series of proof commands that combines several different proof steps. The note command at line 15 first directs Jahob to prove that the receiver is an allocated Hashtable object, and to label this proved lemma ThisProps. Together, the proved lemma ThisProps and the labeled precondition conjunct Init provide the necessary facts to apply *modus ponens* to result of instantiating ContentsDefInv with the receiver object. The next note command, at line 16, directs Jahob to prove that hc is the result of applying the hash function h to k0 and table..length, and to label the proved lemma HCDef. This property is also necessary for applying the ContentsDefInv invariant. Finally, the note command at line 17 directs Jahob to prove that (k0, v0) is in the bucket at index hc, using the facts KeyInContents, ContentsDefInv, Init, ThisProps, and HCDef. From this result, the provers are then able to automatically conclude that f is non-null.

7.6.7 The removeFirst() Method

Figures 7-12 and 7-13 present the body of the removeFirst() method, which is invoked by _remove(). The removeFirst() method removes the binding for a key k0 from the hash table, given that k0 corresponds to the first node in the bucket at index hc. The removeFirst() method first declares a ghost variable v0 and assigns it to the value corresponding to k0 using a havoc command. It also uses the havoc command to give the constraint on v0 the label InContents. Next, removeFirst() removes f—the first node in the bucket at hc—by assigning table[hc] to the next node in the list, and nulling out the next field of the removed node. To restore the hash table invariants, removeFirst() performs two ghost variable assignments. First, it assigns f..con (the contents of the removed node) to the singleton set (f..key, f..value). Then, it assigns the contents of the hash table to the result of removing the pair (k0, v0) from the original value of contents. Finally, at the end of the method, removeFirst() returns f.value—the value to which k0 was previously mapped.

Verifying the Postcondition

The body of removeFirst() contains a number of proof commands that provide the guidance that the provers need to verify the method. The first group of proof commands address the second conjunct in the postcondition—that the pair (k0, result) was originally in content. To prove this property, it is necessary to show that v0 is equal to f..value, which depends on both ContentsDefInv and ConDef. Together, these invariants establish that k0 maps to f..value, and that k0 has only one mapping in the hash table. Since k0 also maps to v0, then v0 and f..value must be the same. To apply ContentsDefInv to the receiver object, the note command in line 22 first directs

```
1  private Object removeFirst(Object k0, int hc)
2  /*: requires "init ∧ k0 ≠ null ∧ (∃v.(k0, v) ∈ contents) ∧
3               comment ''KFound'' (k0 = table.[hc]..key) ∧
4               comment ''HCProps'' (0 ≤ hc ∧ hc < table..length ∧
5               hc = h k0 (table..length)) ∧ theinvs"
6      modifies contents, con, next, arrayState
7      ensures "(contents = old contents - {(k0, result)}) ∧
8               ((k0, result) ∈ old contents) ∧
9               (∀a i.a ≠ table → a.[i] = old (a.[i])) ∧ theinvs" */
10 {
11     //: ghost specvar v0 :: obj;
12     //: havoc v0 suchThat InContents: "(k0, v0) ∈ contents";
13
14     Node f = table[hc];
15     Node second = f.next;
16     f.next = null;
17     //: "f..con" := "{(f..key, f..value)}";
18
19     table[hc] = second;
20     //: "contents" := "old contents - {(k0, v0)}";
21
22     //: note ThisProps: "this ∈ alloc ∧ this ∈ Hashtable ∧ init";
23     /*: note OldContents: "old contents =
24         {(k,v). (k,v) ∈ old (table.[(h k (table..length))]..con)}"
25         from ContentsDefInv, ThisProps; */
26     //: note FNonNull: "f ≠ null";
27     /*: note FProps: "f ∈ Node ∧ f ∈ alloc"
28         from unalloc_lonely, array_pointsto, ThisProps; */
29     /*: note VFound: "v0 = f..value" from InContents, OldContents,
30         ConDef, KFound, FProps, FNonNull, HCProps; */
31
32     /*: note Acyclic: "fieldRead (old next) f ≠ f"
33         from FNonNull, HCProps, FirstInjInv, ThisProps; */
```

Figure 7-12: Body of Hashtable.removeFirst() (continued in Figure 7-13)

```
34      {
35          /*: pickAny ht::obj suchThat
36              ContentsDefHyp: "ht ∈ alloc ∧ ht ∈ Hashtable ∧ ht..init"; */
37          /*: note ContentsThis: "ht = this → ht..contents =
38              {(k,v). (k,v) ∈ ht..table.[(h k (ht..table..length))]..con}"
39              from OldContents, ElementInjInv, Acyclic, ThisProps, KFound,
40              VFound, ConDef, FProps, FNonNull, HashInv, HCProps; */
41          {
42              //: assuming NotThisHyp: "ht ≠ this";
43              /*: note OldHTContents:
44                "fieldRead (old Hashtable.contents) ht =
45                {(k, v). (k, v) ∈ (fieldRead (old con) (arrayRead
46                (old arrayState) (ht..table) (h k (ht..table..length))))}"
47                from ContentsDefHyp, NotThisHyp, ContentsDefInv; */
48              //: note TableNotEq: "ht..table ≠ table";
49              /*: note ContentsOther: "ht..contents = {(k, v).
50                  (k, v) ∈ ht..table.[(h k (ht..table..length))]..con}"
51                  from ContentsDefHyp, NotThisHyp, HashInv, ElementInjInv,
52                  HCProps, ThisProps, FNonNull, OldHTContents,
53                  TableNotEq; */
54          }
55          /*: cases "ht = this", "ht ≠ this" for ContentsCases:
56              "ht..contents = {(k, v).
57               (k, v) ∈ ht..table.[(h k (ht..table..length))]..con}"
58               from ContentsThis, ContentsOther; */
59           /*: note ContentsDefPostCond: "ht..contents = {(k, v).
60              (k, v) ∈ ht..table.[(h k (ht..table..length))]..con}"
61              from ContentCases forSuch ht; */
62      }
63      /*: note CoherencePostCond: "theinv Coherence" from ConDef, ConNull,
64          Coherence, TableInjInv, HCProps, FProps, FNonNull, Acyclic; */
65
66      return f.value;
67  }
```

Figure 7-13: Body of Hashtable.removeFirst() (continued from Figure 7-12)

Jahob to prove that the receiver object is an initialized, allocated Hashtable object and to label this property ThisProps. The next note command, in line 23, then directs Jahob to use the proved lemma ThisProps and the ContentsDefInv invariant to prove that the original value of contents consists of the set of key-value pairs in the original hash table, according to the definition in ContentsDefInv, and to label this property OldContents. The next two note commands establish the facts necessary for applying ConDef to f. The note command at line 26 directs Jahob to prove that f is non-null and to label that property FNonNull. The subsequent note command, at line 27 directs Jahob to prove that f is an allocated Node object, using the proved lemma FProps, and the background lemmas unalloc_lonely and array_pointsto which Jahob provides by default,[1] and to label that property FProps. Finally, the note command at line 29 directs Jahob to use the labeled facts KFound, HCProps, and InContents, the invariant ConDef, and the proved lemmas OldContents, FProps, FNonNull, and HCProps to prove that v0 is equal to f..value, and to give the proved lemma the label VFound.

The ContentsDefInv Postcondition

The next note command in the method (at line 32) directs Jahob to prove an intermediate lemma for use in subsequent proof commands, and to label it Acyclic. This lemma states that the next field of the removed node did not refer to itself in the original state of the hash table. This non-aliasing property is important for showing that the removed node is actually removed from the hash table.

The group of proof commands that follow (starting at line 35 of Figure 7-13) addresses the ContentsDefInv invariant in the method postcondition. Because ContentsDefInv is universally quantified, the proof is enclosed in a pickAny block. This block gives a name (ht) to the quantified variable and introduces the hypothesis that ht conforms to the antecedent in the implication within ContentsDefInv. By naming the quantified variable, the pickAny block simplifies the proof by eliminating the outer universal quantifier from subsequent proof steps. It also gives a label, ContentsDefHyp, to the introduced hypothesis so that later proof commands can refer to it. It now remains to prove the consequent in ContentsDefHyp.

There are two top-level cases in the proof of the consequent: one case considers the receiver object and the other considers all other objects. The first note command in the block (at line 37) addresses the former case. The commands that follow (starting at line 42) address the latter. Re-establishing the ContentsDefInv invariant for the receiver object depends not only on the ContentsDefInv invariant from the method precondition, but also on several other properties including injectivity properties, the definition of con, and properties concerning the index hc and the removed key and value. The note command at line 37 therefore directs Jahob to use the labeled facts HCProps and KFound (from the precondition), the invariants ElementInjInv, ConDef, and HashInv, and the proved lemmas OldContents, Acyclic, ThisProps, VFound, FProps, and FNonNull to prove that ContentsDefInv holds for the receiver object. This single

[1]The lemma unalloc_lonely states the property that newly-allocated objects do not point to any existing objects or vice versa, while array_pointsto provides type information about the objects in an array.

note command is sufficient for guiding the provers to prove this property. The note command also directs Jahob to label this property ContentsThis.

The next part of the proof addresses objects other than the receiver. It begins with an assuming block that introduces the hypothesis (labeled NotThisHyp) that ht is not equal to the receiver. The proof for this case consists of showing that the ContentsDefInv property from the method precondition still holds for ht. Therefore, the first note command in the block, at line 43, directs Jahob to establish the relationship between the original contents of ht and the original concrete state of the hash table. It identifies the necessary supporting facts as the hypotheses ContentsDefHyp and NotThisHyp, and the ContentsDefInv invariant, and directs Jahob to label the proved lemma OldHTContents. The note command at line 48 then directs Jahob to prove that ht..table is not equal to table, and to label the proved lemma TableNotEq. This property distinguishes between the table array for ht from the table array for this and is important for showing that the ht..table is not modified. Finally, the note command at line 49 directs Jahob to prove that ht..contents is equal to $\{(k, v).(k, v) \in$ ht..table.[(h k(ht..table..length))]..con$\}$ for objects other than the receiver. It identifies the supporting facts as the hypotheses ContentsDefHyp and NotThisHyp, the invariants HashInv and ElementInjInv, the precondition conjunct HCProps, and the proved lemmas ThisProps, FNonNull, OldHTContents, and TableNotEq, and directs Jahob to label the proved lemma ContentsOther, closing the assuming block.

The cases command at line 55 then combines the case for the receiver object from ContentsThis and the case for all other objects from ContentsOther to prove that the equivalence holds for any object, labeling the resulting proved lemma ContentsCases. Finally, the note command at line 59 closes the pickAny block, proving from ContentsCases that the ContentsDefInv invariant holds.

The Coherence Postcondition

The last note command in removeFirst() addresses the Coherence invariant. This property depends primarily on the Coherence invariant from the method precondition, but also on the definition of the abstract state component con(since Coherence is defined in terms of con), as well as certain injectivity properties. The note command directs Jahob to prove the Coherence invariant using the labeled fact HCProps, the invariants ConDef, ConNull, Coherence, and TableInjInv, and the proved lemmas FProps, FNonNull, and Acyclic. This single note command is sufficient for providing the guidance needed by the provers to prove the Coherence invariant in the method postcondition.

7.6.8 The removeFromBucket() Method

Figures 7-14, 7-15, 7-16, 7-17, and 7-18 present the body of the removeFromBucket() method. This method takes a key k0 and an index hc and removes the corresponding key-value pair from the bucket at hc. It first declares a ghost variable v0 and uses a havoc command to assign v0 to the value corresponding to k0 in the hash table. It also uses the havoc command to give the constraint on v0 the label InContents. It

```
1  private Object removeFromBucket(Object k0, int hc)
2  /*: requires "comment ''Init'' init ∧ k0 ≠ null ∧
3               (∃v.(k0, v) ∈ contents) ∧
4               comment ''KNotFound'' (k0 ≠ table.[hc]..key) ∧
5               comment ''HCProps'' (0 ≤ hc ∧ hc < table..length ∧
6               hc = h k0 (table..length)) ∧ theinvs"
7      modifies contents, con, next, arrayState
8      ensures "(contents = old contents - {(k0, result)}) ∧
9               ((k0, result) ∈ old contents) ∧
10              (∀a i. a ≠ table → a.[i] = old (a.[i])) ∧ theinvs" */
11 {
12     //: ghost specvar v0 :: obj;
13     //: havoc v0 suchThat InContents: "(k0, v0) ∈ contents";
14
15     Node f = table[hc];
16     Node prev = f;
17
18     /*: note InBucket: "(k0, v0) ∈ prev..con" from InContents,
19         ContentsDefInv, thisNotNull, thisType, Init, HCProps; */
20     /*: note PrevNotNull: "prev ≠ null"
21         from InBucket, ConDef, ConNull; */
22
23     //: "prev..con" := "prev..con - {(k0, v0)}";
24     //: "contents"   := "old contents - {(k0, v0)}";
25
26     Node curr = prev.next;
27
28     /*: note PrevHidden: "prev ∈ hidden" from NodeHidden1,
29         thisNotNull, thisType, PrevNotNull, Init, HCProps; */
30
31     /*: note ConPreLoop:
32         "∀n. n ∈ Node ∧ n ∈ alloc ∧ n ≠ null ∧ n ≠ prev →
33          n..con = {(n..key, n..value)} ∪ n..next..con ∧
34          (∀v.(n..key, v) ∉ n..next..con)"
35         from ConDef, FirstInjInv, Init, HCProps, thisNotNull,
36         thisType, PrevNotNull; */
37
38     /*: note ConUnchangedPreLoop:
39         "∀ht i. ht ≠ this ∧ ht ∈ Hashtable ∧ ht ∈ alloc ∧ ht..init ∧
40          0 ≤ i ∧ i < ht..table..length →
41          ht..table.[i]..con = old (ht..table.[i]..con)"
42         from ElementInjInv, thisType, PrevNotNull, Init, HCProps; */
```

Figure 7-14: Body of Hashtable.removeFromBucket() (continued in Figure 7-15)

```
43      while /*: inv "prev ∈ Node ∧ prev ∈ alloc ∧ prev ≠ null ∧
44                     prev ∈ hidden ∧ comment ''PrevCon''
45                     (prev..con = fieldRead (old con) prev - {(k0, v0)}) ∧
46                     comment ''PrevNot''
47                     (∀v.(prev..key, v) ∉ prev..next..con) ∧
48                     comment ''CurrProps'' (curr ∈ Node ∧ curr ∈ alloc) ∧
49                     comment ''CurrNotNull'' (curr ≠ null) ∧
50                     comment ''PrevCurr''
51                     (prev..next = curr ∧ prev ≠ curr) ∧
52                     contents = old contents - {(k0, v0)} ∧
53                     (k0, v0) ∈ curr..con ∧ comment ''ConDefInv''
54                     (∀ n. n ∈ Node ∧ n ∈ alloc ∧ n ≠ null ∧ n ≠ prev →
55                     n..con = {(n..key, n..value)} ∪ n..next..con ∧
56                     (∀v.(n..key, v) ∉ n..next..con)) ∧
57                     comment ''ConLoop''
58                     (∀n. n..con = old (n..con) ∨
59                     n..con = old (n..con) - {(k0, v0)}) ∧
60                     (null..con = ∅) ∧ comment ''FConInv''
61                     (f..con = (fieldRead (old con) f) - {(k0, v0)}) ∧
62                     comment ''ConUnchanged''
63                     (∀ht i. ht ≠ this ∧ ht ∈ Hashtable ∧ ht ∈ alloc ∧
64                     ht..init ∧ 0 ≤ i ∧ i < ht..table..length →
65                     ht..table.[i]..con = old (ht..table.[i]..con))" */
66          (curr.key != k0)
67      {
68          //: "curr..con" := "curr..con - {(k0, v0)}";
69
70          /*: note CurrCon:
71              "curr..con = fieldRead (old con) curr - {(k0, v0)}"; */
72          //: note PrevIsNot: "prev..key ≠ k0";
73          /*: note OldConDef: "fieldRead (old con) prev =
74              (prev..key, prev..value)} ∪
75              fieldRead (old con) (prev..next)"; */
76          /*: note PrevConDef:
77              "prev..con = {(prev..key, prev..value)} ∪ prev..next..con"
78              from PrevCurr, PrevCon, CurrCon, OldConDef, PrevIsNot; */
79
80          prev = curr;
81          curr = curr.next;
82
83          /*: note FConLem:
84              "f..con = (fieldRead (old con) f) - {(k0, v0)}"
85              from FConInv; */
86
87          /*: note ConExceptPrev:
88              "∀n. n ∈ Node ∧ n ∈ alloc ∧ n ≠ null ∧ n ≠ prev →
89               n..con = {(n..key, n..value)} ∪ n..next..con ∧
90               (∀v.(n..key, v) ∉ n..next..con)" from PrevConDef,
91              PrevNot, ConDefInv, PrevCurr, NextInjInv, CurrNotNull; */
92      }
```

Figure 7-15: Body of Hashtable.removeFromBucket() (continued in Figure 7-16)

```
93      Node tmp = curr.next;
94      prev.next = tmp;
95      curr.next = null;
96
97      //: "curr..con" := "{(curr..key, curr..value)}";
98
99      {
100         /*: pickAny x::obj suchThat
101             xHyp: "x ∈ Node ∧ x ∈ alloc ∧ x ≠ null"; */
102         {
103             //: assuming xIsPrev: "x = prev";
104             /*: note nextNotCurr: "fieldRead (old next) curr ≠ curr"
105                 from NextInjInv, CurrNotNull, PrevCurr, CurrProps; */
106             /*: note prevNextCon:
107                 "prev..next..con = fieldRead (old con) (prev..next)"; */
108             /*: note prevOldCon: "fieldRead (old con) prev =
109                 {(prev..key, prev..value)} ∪
110                 fieldRead (old con) curr"; */
111             /*: note currOldCon: "fieldRead (old con) curr =
112                 {(curr..key, curr..value)} ∪
113                 fieldRead (old con) (fieldRead (old next) curr)"; */
114             //: note prevKeyNotK0: "prev..key ≠ k0";
115             {
116                 /*: pickAny k::obj, v::obj suchThat
117                     ForwHyp: "(k, v) ∈ x..con"; */
118                 //: note kNotK0: "k ≠ k0";
119                 //: note currKeyIsK0: "curr..key = k0";
120                 /*: note ForwCase:
121                     "(k, v) ∈ {(x..key, x..value)} ∪ x..next..con"
122                     from xHyp, xIsPrev, ForwHyp, PrevCurr, nextNotCurr,
123                     PrevCon, prevNextCon, prevOldCon, currOldCon,
124                     prevKeyNotK0, kNotK0, currKeyIsK0 forSuch k, v; */
125             }
126             /*: note BackCase:
127               "∀k v.(k, v) ∈ {(x..key, x..value)} ∪ x..next..con →
128                (k, v) ∈ x..con"; */
129             /*: note xCon: "x..con =
130                 {(x..key, x..value)} ∪ x..next..con"
131                 from ForwCase, BackCase; */
132         }
133         /*: cases "x = curr", "x = prev", "x ≠ curr ∧ x ≠ prev" for
134             XCon: "x..con = {(x..key, x..value)} ∪ x..next..con"; */
135         /*: note ConPost:
136             "x..con = {(x..key, x..value)} ∪ x..next..con ∧
137              (∀v.(x..key, v) ∉ x..next..con)" forSuch x; */
138     }
```

Figure 7-16: Body of Hashtable.removeFromBucket() (continued in Figure 7-17)

```
139     {
140         /*: pickAny ht::obj suchThat
141             CohHyp: "ht ∈ alloc ∧ ht ∈ Hashtable ∧ ht..init"; */
142         {
143             /*: pickAny i::int, k::obj, v::obj suchThat
144                 InnerHyp: "0 ≤ i ∧ i < ht..table..length ∧
145                 (k, v) ∈ ht..table.[i]..con"; */
146             //: note NotCurr: "ht..table.[i] ≠ curr";
147             /*: note InnerConc: "h k (ht..table..length) = i"
148                 from CohHyp, InnerHyp, Coherence, NotCurr, ConLoop
149                 forSuch i, k, v; */
150         }
151         /*: note CoherencePost:
152             "(∀i k v.0 ≤ i ∧ i < ht..table..length →
153              (k, v) ∈ ht..table.[i]..con → h k (ht..table..length) = i)"
154              from InnerConc forSuch ht; */
155     }
156     {
157         /*: pickAny x::obj suchThat
158             ContentsDefHyp: "x ∈ alloc ∧ x ∈ Hashtable ∧ x..init"; */
159         /*: note OldXContents: "fieldRead (old Hashtable.contents) x =
160             {(k, v). (k, v) ∈ (fieldRead (old con) (arrayRead
161             (old arrayState) (x..table) (h k (x..table..length))))}"
162             from ContentsDefHyp, ContentsDefInv; */
163         {
164             //: assuming XNotThisHyp: "x ≠ this";
165             /*: note NotCurr: "∀i.0 ≤ i ∧ i < x..table..length →
166                 x..table.[i] ≠ curr"; */
167             /*: note ConXUnchanged:
168                 "∀i.0 ≤ i ∧ i < x..table..length →
169                  x..table.[i]..con = fieldRead (old con)
170                  (arrayRead (old arrayState) (x..table) i)" from
171                 XNotThisHyp, ContentsDefHyp, NotCurr, ConUnchanged; */
172             /*: note LengthLemma: "∀k.0 ≤ (h k (x..table..length)) ∧
173                 (h k (x..table..length)) < (x..table..length)"
174                 from ContentsDefHyp, HashInv; */
175             /*: note XNotThisCase: "x..contents = {(k, v).
176                 (k, v) ∈ x..table.[(h k (x..table..length))]..con}"
177                 from ContentsDefHyp, XNotThisHyp, OldXContents,
178                 LengthLemma, ConXUnchanged; */
179         }
```

Figure 7-17: Body of Hashtable.removeFromBucket() (continued in Figure 7-18)

```
180         {
181             //: assuming XIsThisHyp: "x = this";
182             /*: note OldContents: "old contents = {(k,v).
183                 (k,v) ∈ old (table.[(h k (table..length))]..con)}"; */
184             {
185                 /*: pickAny k::obj, v::obj suchThat
186                     ForwHyp: "(k, v) ∈ contents"; */
187                 /*: note NotCurr:
188                     "table.[(h k (table..length))] ≠ curr" from
189                     FirstInjInv, ContentsDefHyp, XIsThisHyp, PrevCurr,
190                     CurrNotNull, HashInv; */
191                 /*: note ForwCase:
192                     "(k, v) ∈ table.[(h k (table..length))]..con"
193                     from ForwHyp, OldContents, NotCurr, ConLoop
194                     forSuch k, v; */
195             }
196             {
197                 /*: pickAny k::obj, v::obj suchThat BackHyp:
198                     "(k, v) ∈ table.[(h k (table..length))]..con"; */
199                 /*: note NotCurr: "table.[(h k (table..length))] ≠ curr"
200                     from FirstInjInv, ContentsDefHyp, XIsThisHyp,
201                     PrevCurr, CurrNotNull, HashInv; */
202                 /*: note BackCase: "(k, v) ∈ contents" from BackHyp,
203                     OldContents, NotCurr, ConLoop, FConInv, HCProps
204                     forSuch k, v; */
205             }
206             /*: note XIsThisCase: "contents =
207                 {(k, v). (k, v) ∈ table.[(h k (table..length))]..con}"
208                 from ForwCase, BackCase; */
209         }
210         /*: note ContentsDefPost: "x..contents =
211             {(k, v). (k, v) ∈ x..table.[(h k (x..table..length))]..con}"
212             from XNotThisCase, XIsThisCase forSuch x; */
213     }
214     return curr.value;
215
216 }
```

Figure 7-18: Body of Hashtable.removeFromBucket() (continued from Figure 7-17)

then traverses the the bucket at hc using a while loop, starting with the second node in the linked list. (The method precondition guarantees that the entry for k0 is not in the first node of the bucket.) In each iteration of the loop, a specification assignment statement removes the pair (k0, v0) from curr..con, where curr is the current node in the traversal. The loop also keeps track of the previous node on the list. The loop exits when it reaches the node corresponding to k0. At this point, the method removes the node corresponding to k0 by assigning the next field of the previous node on the list to curr.next, and assigns curr.next to null. A specification assignment statement assigns the value of curr..con to the singleton set consisting of the key-value pair at the current node, restoring the ConDef invariant. Finally, the method returns curr.value—the value that was previously associated with k0 in the hash table.

Note that our implementation of Hashtable.remove() suffers from the same inefficiency as AssociationList.remove(), since both _containsKey() and _remove() must search the hash table for the removed key. As in the case of the association list, this simplification results in smaller methods, which generally corresponds to a simpler verification. Even so, the verification for the methods involved, which include removeFromBucket(), is non-trivial. While it is also possible to verify a more efficient version of Hashtable.remove(), the verification may be more difficult, and may involve the specification of additional abstract state.

Null Dereference Check

The body of removeFromBucket() contains a number of proof commands that provide the guidance the provers need to verify the method. The first pair of note commands (at lines 18 and 20) addresses the null dereference check for the dereference of prev at line 26. Without the guidance of these proof commands, the provers are unable to statically verify that prev is not null within Jahob's standard time limit. The value of prev is guaranteed to be non-null because k0 is known to be in the hash table, at the bucket that begins with the node prev. In addition to InContents, which indicates that k0 is in the hash table, the proof also requires the definition of contents from ContentsDefInv, and to apply that definition, certain properties of the receiver object, as well as the definition of the index hc in terms of the hash function h. The first note command therefore directs Jahob to prove that the pair (k0, v0) is in prev..con using the constraint InContents, the invariant ContentsDefInv, the background lemmas thisNotNull and thisType, which Jahob provides by default, and the conjuncts Init and HCProps from the precondition. (The background lemma thisNotNull simply states that the receiver this is not equal to null, while thisType gives the type of this, in this case Hashtable, and states that this is allocated.) It also directs Jahob to label the proved lemma InBucket. To prove from this result that prev is non-null requires the definition of con. The next note command therefore directs Jahob to prove that prev is not equal to null using the proved lemma InBucket, and the invariants ConDef and ConNull. It directs Jahob to label the proved lemma PrevNotNull so it can be used in subsequent proof commands.

Loop Invariant Base Case

The next set of note commands (at lines 28, 31, and 38) guides the system in proving that the loop invariant holds on the initial entry into the loop. Each command identifies the set of facts to use to prove a particular conjunct in the loop invariant. The first note command addresses the loop invariant property prev $\in$ hidden. This property becomes important at the end of the loop, when the next field of prev is assigned, to ensure that references to hidden objects are only written in other hidden objects. The NodeHidden1 invariant from the precondition ensures this property for the first node in every bucket of the hash table. But the proof also requires properties of the receiver object and of the index hc to apply the invariant to prev. The note command therefore directs Jahob to prove this property using the NodeHidden1 invariant, thisNotNull, thisType, PrevNotNull, Init, and HCProps.

The second note command in this set, at line 31, addresses the conjunct in the loop invariant with the label ConDefInv, which is later used in proving that ConDef is restored at the end of the method. This conjunct indicates that the ConDef invariant holds at the top of the loop for all nodes except prev. This is because the loop modifies con of the current node curr by removing the pair (k0, v0), temporarily violating the relationship between the current node and its successor given by ConDef. This relationship is restored at the next iteration of the loop, when con of the successor node is modified. The proof of the desired property depends primarily on ConDef, but also on FirstInjInv, to show that prev is not the successor to any node. The note command at line 31 therefore directs Jahob to prove ConDefInv using the invariants ConDef and FirstInjInv, Init, HCProps, thisNotNull, thisType, and PrevNotNull.

The third note command in this set, at line 38, addresses the loop invariant conjunct with the label ConUnchanged. This conjunct indicates that for all allocated and initialized hash tables except for the receiver object, the contents of all the buckets—as defined by the con field of the first node in the bucket—are unchanged from the beginning of the method. The proof of this property depends on ElementInjInv, which indicates that the first nodes of the buckets of one hash table are not the same as the first nodes of the buckets of another. Certain properties of the receiver object, prev, and the index hc are also needed to apply this invariant. The third note command at line 38 therefore directs Jahob to prove this conjunct using ElementInjInv, thisType, PrevNotNull, Init, and HCProps.

Loop Invariant Inductive Case

The proof commands within the loop body guide the provers in the inductive proof of the loop invariant. The goal is to show that the loop invariant holds at the end of the loop body, given the assumption that the loop invariant holds before the loop test. The note commands at lines 70, 72, 73, 76, and 87 address the ConDefInv conjunct of the loop invariant. Earlier proof commands guide the provers in proving this conjunct for the initial entry into the loop. Here, the proof commands guide the provers in proving that this conjunct holds after every loop iteration. The proof consists of showing that the modification of con in the current loop iteration restores

the definition of con for prev, but does not violate it for any node other than curr. The properties required for this proof therefore include the value of con for prev. The next three note commands identify intermediate lemmas needed to derive this value. The first note command (at line 70) concerns the value of con for curr. It directs Jahob to prove that curr..con is the result of removing the pair (k0, v0) from its original value, and labels the proved lemma CurrCon. The next note command (at line 72) directs Jahob to prove that the key associated with the previously traversed node is not k0, and to label it PrevIsNot. The note command at line 73 concerns the original value of con for prev. It directs Jahob to prove that this value consists of the key, value pair at prev and the original value of con for the node following prev in the linked list. The note command at line 76 combines these three proved lemmas as well as the loop invariant conjuncts PrevCurr and PrevCon to show that the value of con for prev is consistent with the definition given by ConDef—i.e., it consists of the key-value pair at prev and the value of con for the next node in the linked list. Finally, the note at line 87 directs Jahob to prove ConDefInv using the proved lemma PrevConDef, and the loop invariant conjuncts ConDefInv, PrevCurr, CurrNotNull, and the invariant NextInjInv. PrevConDef and PrevNot give the definition of con for prev, while ConDefInv gives the definition for the unmodified nodes. The NextInjInv invariant from the precondition guarantees that no other nodes are affected, while PrevCurr and CurrNotNull provide properties necessary to apply the invariant.

The remaining note command in the loop body, at line 83, identifies the fact that Jahob should use to prove that the loop invariant conjunct FConInv holds at the end of the loop body. In this case, the only necessary fact is the same conjunct from the inductive hypothesis for the loop invariant indicating that FConInv holds on entry to the loop. By identifying this loop invariant conjunct as the only necessary fact for the proof, this note command enables the provers to prove the desired property. Without a note command to identify this fact, the provers are unable to find the proof within Jahob's standard time limit due to the large number of facts in the assumption base. All other conjuncts of the loop invariant prove without additional proof commands.

The ConDef Postcondition

The proof commands at the end of the method address conjuncts in the method postcondition. The pickAny block at line 100 guides Jahob in proving that the ConDef invariant is restored at the end of the method. The pickAny command names the quantified variable in ConDef, making it possible to state intermediate lemmas that reference that variable. It also hypothesizes that the antecedent of the implication within the universal quantifier holds. It gives the name x to the quantified variable and labels the hypothesis xHyp. It now remains to prove the antecedent, which has two parts. The first part specifies that x..con consists of the key-value pair at x and the value of con of the following node in the linked list. The second part specifies the key at x is not present in the rest of the linked list. There are three main cases in the proof of the first part: the case for x equal to curr, the case for x equal to prev, and the case for all other values of x. The first and third cases prove without proof commands, but the case for prev requires some guidance.

The assuming block at line 103 addresses the case for prev. It introduces the hypothesis that x is equal to prev, the node preceding the removed node curr, and gives the label xIsPrev to the hypothesis. The proof shows that x..con is equal to $\{x..key, x..value\} \cup x..next..con$ by showing that the two sets have the same contents.

The first set of note commands direct Jahob to prove some intermediate lemmas used later in the proof. The note command at line 104 directs Jahob to prove that the next field of curr in the original hash table did not point to itself, and to label the proved lemma nextNotCurr. It identifies the facts to use as the NextInjInv invariant, and the conjuncts CurrNotNull, PrevCurr, and CurrProps from the loop invariant. The proved lemma states an injectivity property necessary for showing that the removed node is actually removed.

The next note command, at line 106, directs Jahob to prove that the con field of prev..next is the same as it was in the original hash table, and to label the proved lemma prevNextCon. This property gives the value of con for prev..next, which is important for showing that the definition of con for prev is restored.

The note command at line 108 concerns the original value of con for prev. It directs Jahob to prove that the con field of prev in the original hash table is given by the definition in ConDef, and to label the proved lemma currOldCon. The following note command, at line 111, directs Jahob to prove the same property for curr, and to label it currOldCon. The note command at line 114 directs Jahob to prove that prev..key is not equal to k0 and to label the proved lemma prevKeyNotK0.

The pickAny block at line 116 addresses the forward direction of the proof for showing that the sets x..con and $\{x..key, x..value\} \cup x..next..con$ are equal. The pickAny command chooses arbitrary k and v and considers the hypothesis that the pair (k, v) is in x..con. It labels this hypothesis ForwHyp. The goal is to show that (k, v) is also in $\{x..key, x..value\} \cup x..next..con$. The proof depends primarily on the definition of con for prev as given by the loop invariant conjunct PrevCon, but also on the values given by the previously proved lemmas, and the relationships between prev, curr, k0, and k as expressed by various proved lemmas and loop invariant conjuncts. The next two note commands direct Jahob to prove two of the intermediate lemmas that are necessary for the proof. The note command at line 114 directs Jahob to prove that k is not equal to k0 and to label the proved lemma kNotK0. The note command at line 118 directs Jahob to prove that curr..key is equal to k0, and to label the proved lemma currKeyIsK0. The note command at line 120 then directs Jahob to use the hypotheses xHyp, kIsPrev and ForwHyp, the loop invariant conjuncts PrevCurr and PrevCon, and the proved lemmas nextNotCurr, prevNextCon, prevOldCon, CurrOldCon, prevKeyNotK0, kNotK0, currKeyIsK0 to prove that (k, v) is in $\{(x..key, x..value)\} \cup$ x..next..con for the given k and v. The loop invariant conjunct PrevCurr provides the definition of con for prev, while PrevCurr and nextNotCurr give the relationship between curr and prev. The proved lemmas prevNextCon, prevOldCon, and CurrOldCon provide other definitions of con necessary for the proof, while prevKeyNotK0, kNotK0, and currKeyIsK0 give the relationship between k0 and other keys. This note command closes the pickAny block and labels the proved lemma ForwCase. It completes the forward direction of the proof by showing that an arbitrary pair (k, v) in x..con must also be in $\{x..key, x..value\} \cup x..next..con$.

The note command at line 126 then directs Jahob to prove the converse of ForwCase and to label it BackCase. This proof requires only the single note command, completing the backward direction of the proof. The note command at line 129 then directs Jahob to use ForwCase and BackCase to prove the desired equivalence—that x..con consists of the key-value pair at x and the con field of the following node when x is equal to prev. This note command completes the proof of the first part of ConDef for prev, and closes the assuming block.

The cases command at line 133 then directs Jahob to prove the first part of ConDef for all three cases using case analysis. The case for prev is already proved, while the other two cases prove without additional proof commands. The final note command at line 135 closes the pickAny block by directing Jahob to conclude both parts of the ConDef invariant for the given x. The first part is proved in the previous cases command, while the second part proves without additional proof commands, completing the proof of the ConDef invariant in the postcondition.

The Coherence Postcondition

The pickAny block that begins at line 140 guides Jahob in proving the Coherence invariant. Without it, the provers are unable to prove the Coherence component of the postcondition within the standard time limit. The outer pickAny block names the quantified variable at the top level of the Coherence invariant and makes it possible to state intermediate lemmas that reference that variable. It gives the variable the name ht and hypothesizes that the antecedent of the top-level implication in Coherence holds for ht. Specifically, the hypothesis indicates that ht is an allocated, initialized Hashtable object. The pickAny command labels this hypothesis CohHyp.

The outer pickAny block also encloses an inner pickAny block that names the variables quantified by the inner universal quantifier of Coherence. The inner pickAny block names these variables i, k, and v. It also hypothesizes that the antecedent of the implication in this inner quantifier holds and names the hypothesis InnerHyp. This hypothesis indicates that i must be in the range $[0, \mathsf{ht..table..length})$ and that the pair (k, v) is in the ith bucket of ht, or, more precisely, that it is in ht..table.[i]..con.

It now remains to prove the consequent of the implication—i.e., that the ith bucket is the correct bucket for k as defined by the hash function h. The proof of this property depends on the Coherence invariant in the precondition and on the loop invariant. The loop invariant conjunct ConLoop expresses the property that the abstract state component con for all objects is either unchanged or the result of removing the pair $(\mathsf{k0}, \mathsf{v0})$ from the original value of con. Therefore, at the end of the loop, any pair (k, v) in the ith bucket of ht must also have been in the ith bucket of ht in the original state. According to the Coherence precondition, any such pair is in correct bucket at the end of the while loop. But between the end of the loop and the current program point, the value of con has changed for the removed node curr. Therefore, it is important to distinguish curr from the node corresponding to the head of the bucket that contains k. The note command at line 146 does this by directing Jahob to prove an intermediate lemma—that the first node in the ith bucket in ht is not the removed node curr. This proved lemma, along with the Coherence invariant, the hypotheses CohHyp and

InnerHyp (which are necessary for applying Coherence to the named variables ht, i, k, and v), and the ConLoop conjunct from the loop invariant form a sufficient basis for the provers to show that the Coherence invariant holds. The note command at line 147 directs Jahob to prove the consequent in the inner implication of Coherence using the above facts, thereby establishing the inner universally quantified subformula, and closing the corresponding pickAny block. The note command at line 151 directs Jahob to use the proved result to close the outer pickAny block, successfully proving the Coherence postcondition, which Jahob can then add to the assumption base.

The ContentsDefInv Postcondition

The pickAny block that starts at line 157 guides Jahob in proving the ContentsDefInv invariant. Without it, the provers are unable to prove this component of the postcondition within the standard time limit. The proof considers two cases—the receiver object, and all other objects. The pickAny block gives the name x to the quantified variable in ContentsDefInv, and considers the hypothesis that x is an allocated, initialized Hashtable object. It gives the label ContentsDefHyp to this hypothesis, which constitutes the antecedent of the implication in the ContentsDefInv invariant. It now remains to prove the consequent of the implication, namely, that the contents of x is equal to the set of all pairs found in the abstract state component con for the buckets in x..table, given the constraint that each key is mapped to the appropriate bucket in accordance with the hash function h. The proof of this property depends primarily on the ContentsDefInv component of the precondition. The note command at line 159 therefore establishes the value of contents for x in the original state, using ContentsDefInv and ContentsDefHyp (which is necessary for applying ContentsDefInv). It labels the proved lemma OldXContents.

The first assuming block addresses the case for objects other than the receiver. It considers the hypothesis that x is not equal to this, and gives the label XNotThisHyp to the hypothesis. The goal of this case of the proof is to show that the ContentsDefInv invariant from the precondition still applies. The ConUnchanged conjunct of the loop invariant already establishes that the abstract component con is unchanged from before the method executed for the buckets of all hash tables other than the receiver. But because the con component for the removed node curr is modified between the end of the loop and the current program point, it is necessary to establish that curr is not the first node in the linked list of any bucket of x. The note command at line 165 directs Jahob to prove this property and to label the proved lemma NotCurr. The next note command, at line 167, directs Jahob to use NotCurr, ConUnchanged, ContentsDefHyp, and XNotThisHyp, to prove that the contents of the buckets of x are the same as the contents from the beginning of the method. It also gives the label ConXUnchanged to the proved lemma. To apply this lemma, which constrains only valid indices of the hash table, it is necessary to show that the hash function produces valid indices. The HashInv invariant from the precondition provides this property. The note command at line 172 directs Jahob to prove this property from HashInv and ContentsDefHyp, which is necessary to apply HashInv. The note command at line 175 then combines OldXContents, ConXUnchanged, LengthLemma, and XNotThisHyp

to prove the desired consequent for all objects other than the receiver. It labels this proved lemma XNotThisCase.

The second assuming block addresses the case of the receiver object. It considers the hypothesis that x is equal to the this, and gives the label XIsThisHyp to the hypothesis. The note command at line 182 directs Jahob to establish the value of contents for this in the original state, and to label the proved lemma OldContents. This property is very similar to OldXContents, but is made specific to this for use in this case of the proof. Because the concrete and abstract state for the receiver object is actually modified by the method, the proof for this case is more difficult than for the previous case. To prove the equivalence between the set contents and its definition in terms of the combined contents of the buckets, the proof commands first address the case of an arbitrary pair (k, v) in contents, and proves that the same pair is in the set given by the definition. They then prove the converse. The first pickAny block considers the former case by naming the variables k, v, and considering the hypothesis that the pair (k, v) is in contents. It gives this hypothesis the label ForwHyp. The proof for this case depends primarily on OldContents, and on the loop invariant conjunct ConLoop, which defines how the value of con has changed with respect to its value at the beginning of the method. But because the value of con has changed for the removed node curr since the end of the loop, it is again important to establish that none of the nodes under consideration is equal to curr. The note command at line 187 directs Jahob to prove this property using FirstInjInv, HashInv, ContentsDefHyp, XIsThisHyp, and the loop invariant conjuncts PrevCurr and CurrNotNull, and to label the proved lemma NotCurr. The following note command, at line 191, then directs Jahob to use this proved lemma, the hypothesis ForwHyp, OldContents, and ConLoop to prove that (k, v) is in table.[(h k (table.length))]..con for all (k, v) in contents. It gives the proved lemma, which is quantified over k and v, the label ForwCase.

The second case for the receiver addresses the converse of ForwCase, and considers arbitrary k and v such that (k, v) is in table.[(h k (table.length))]..con. The pickAny command at line 197 labels this hypothesis BackHyp. It is again necessary, and for the same reason, to show that the nodes under consideration are not equal to curr. The note command at line 199, which is identical to the one at line 182, including the label NotCurr, directs Jahob to prove this property. (The only difference between the two note commands is that the variables k and v refer to different variables local to their respective pickAny blocks.) The following note command, at line 202 then directs Jahob to use NotCurr, BackHyp, OldContents, ConLoop, FConInv, and HCProps to prove that (k, v) is in contents for all (k, v) in table.[(h k (table.length))]..con. In contrast to the note command at line 191, this note command uses the additional facts FConInv from the loop invariant, and HCProps from the method precondition. The former indicates that the pair (k0, v0) was indeed removed from the bucket at hc, while the latter gives the definition of hc in terms of the hash function h. These two properties are needed to show that the removed pair (k0, v0) is not present in any bucket. However, they were not necessary in the note command at line 191, because the hypothesis ForwHyp constrained k and v such that they could not be equal to k0 and v0.

With both the forward and backward cases proved, the note command at line 167

combines BackCase and ForwCase to obtain the desired equivalence for this and to close the second assuming block. It also gives the label XIsThisCase to the proved lemma. The final note command, at line 210 combines XNotThisCase and XIsThisCase to complete the proof for the ContentsDefInv invariant in the postcondition.

7.7 Discussion

Although the verification for removeFromBucket() seems very involved, there are, in fact, only eight properties that need to be proved using proof commands. Also, many of the proof commands (in particular, pickAny and assuming commands) follow directly from the structure of the goal formula. For example, in the proof of the Coherence invariant for the postcondition, four of the five proof commands follow directly from the structure of the invariant. The remaining command identifies an intermediate lemma needed for the proof, stating that the removed node is not the first node of any bucket in any hash table.

In general, many of the intermediate lemmas used in the Hashtable verification involve the non-aliasing of modified and unmodified state. Other common patterns for intermediate lemmas include: 1) frame conditions identifying unmodified state, 2) lemmas that capture the program state before any modifications, and 3) instantiations of invariants for either the receiver object, or for a particular quantified object. These lemmas are used to distinguish between potentially aliased program state, and to re-establish partially violated invariants for both modified and unmodified objects. The need for these lemmas support the hypothesis that managing aliased state is one of the main sources of complexity in verifying imperative data structures. This complexity is particularly noticeable in the verification of instantiable data structures, like the hash table, due to the additional layer of potential aliasing.

Of our verified data structures, the hash table was one of the most difficult to verify. The verification effort was slighter greater than that of the priority queue. As with the priority queue, part of the complexity stems from the combination of structural and arithmetic properties, which results in the need for proof commands to coordinate the efforts of multiple provers. Another source of complexity is the large number of class invariants. Due to its inherent complexity, the hash table contains more invariants than any of the other data structures. These additional invariants result in an increase in the number of assumptions in the assumption base, and a corresponding increase in the size of the proof search space for the generated sequents. Consequently, there is a greater need in the hash table verification for note commands with from clauses. These note commands appropriately limit the assumption base to the relevant assumptions, enabling the provers to successfully prove the generated verification conditions.

Note that the verification effort is surprisingly low for methods that indirectly manipulate the hash table state, like remove() and _remove(). Most of the proof commands used in the verification occur in methods that directly modify the concrete state of the hash table. Methods that indirectly modify the hash table state—by invoking other methods—require few or no proof commands. The main verification

burden appears to lie in establishing the correspondence between the concrete and abstract state. Once established for the called methods, the verification for the callers involved only the sequential composition of updates to the abstract state, which was much easier to verify. This result suggests that data structure clients, which manipulate data structures indirectly through called methods, may be substantially easier to verify than data structure implementations, which directly modify the data structure state.

7.8 Summary

This chapter describes the specification and verification of a hash table data structure. Our hash table is implemented using an array with separate chaining, where the contents of the buckets are stored in singly-linked lists. Its abstract state is modeled as a set of key-value pairs. The hash table maintains invariants that guarantee the correct hashing of key-value pairs in the hash table, and the absence of duplicate mappings for keys. The public operations that it supports include the standard put(), remove(), and containsKey() operations, as well as operations like replace(), that export a slightly different interface.

The hash table data structure was one of the most difficult data structures to verify. This difficulty is partly due to the combination of structural and arithmetic properties in the hash table. This combination required proof commands to coordinate multiple provers in the proof of the different types of properties. The hash table verification was also made more difficult by the large number of class invariants needed to specify the properties that it maintains. These invariants resulted in a larger assumption base, and consequently a larger proof search space. It was therefore necessary to use many note commands with from clauses to appropriately limit the assumption base to enable the provers to successfully prove the generated sequents.

We found it useful to implement the functionality of the hash table in many small methods, which were easier to verify than larger methods. Top-level methods invoked private methods that implemented the actual functionality. These top-level methods, which indirectly modify the hash table state through called methods, were generally easy to verify, requiring few or no proof commands. The majority of the proof commands occur in methods that directly modify the hash table state, due to the difficulty of establishing the correspondence between updates to the concrete and abstract states. Verification of the callers, which involved only the sequential composition of updates to the abstract state, was much easier, and, in some cases, trivial. This result suggests that data structure clients, which manipulate the data structure state indirectly through called methods, may be substantially easier to verify than data structure implementations. In that case, the effort of verifying data structure libraries would not only be amortized over many uses, but would comprise a significant portion of the effort required to verify larger programs that use these libraries, further supporting the notion of verified data structure libraries.

Chapter 8

Experimental Results

This chapter discusses the empirical results that we obtained from the data structure implementations that we verified and the observations that we made based on those results. We used Jahob to specify and verify a range of recursive and array-based imperative linked data structures, including lists, trees, and hash tables. We verified these data structures with respect to various specifications that capture set and map abstractions. Our results support our integrated reasoning approach—each data structure required more than one prover to verify. (The maximum number of different provers used to verify a single data structure was six.) The integrated proof language was also essential to the success of the verification. The amount of developer guidance needed for each data structure (in the form of proof commands) spanned a wide range. The simplest data structures—the singly-linked list and the association list described in Chapter 6—required no guidance. The most difficult data structures—the priority queue and hash table described in Chapters 5 and 7, respectively—required a much more substantial effort to verify. This effort consists primarily of note commands, both with and without from clauses. However, the verification also required the use of many other proof commands, including assuming and pickAny. Although, in some cases, many proof commands were required for a successful verification, these proof commands were focused on a small number of difficult formulas. The overwhelming majority of the formulas generated by the system were verified automatically by the combined reasoning system.

8.1 Verified Data Structures

We measured the results described in this chapter for the following data structures, which we specified and verified using Jahob. With the exception of the binary search tree and circular list, all of the following verified data structures are instantiable.

- **Array List:** A list stored in an array implementing a map from integers to objects, optimized for storing maps from a dense subset of the integers starting at 0 (modeled after java.util.ArrayList). Method contracts in the list describe operations using an abstract relation $\{(0, v_0), \ldots, (k, v_k)\}$, where $k+1$ is the number of stored elements. In addition to the standard list operations for observing

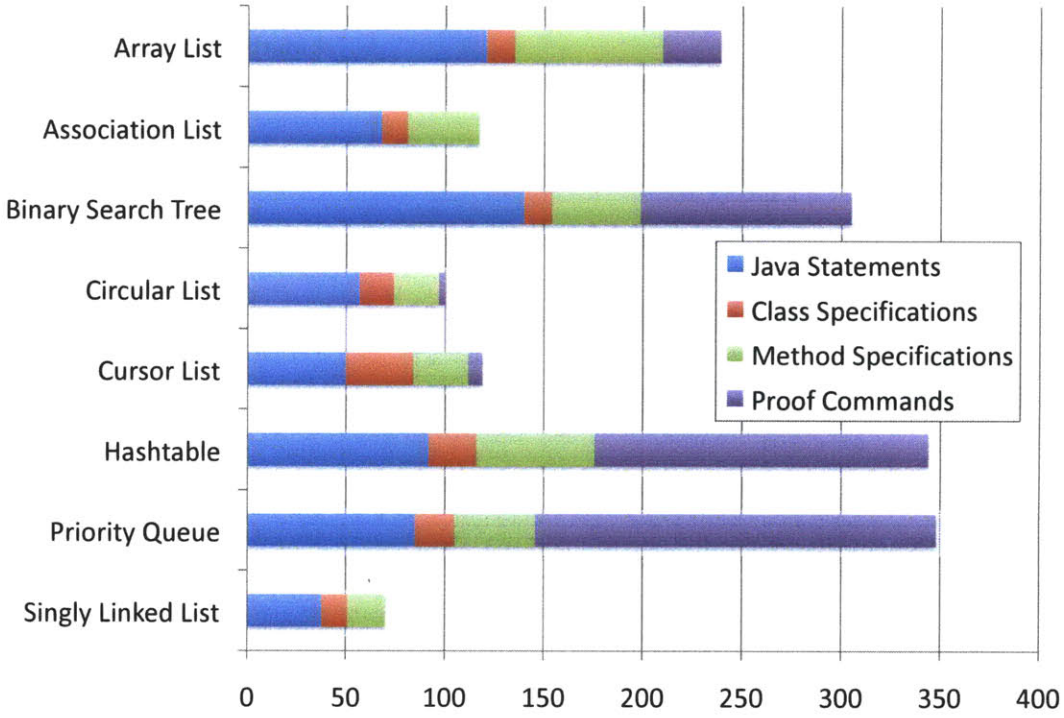
and modifying membership, the implementation also supports operations specific to array lists, including trimToSize(), which shrinks the capacity of the list to the minimum needed for the currently contained elements, ensureCapacity(), which increases the capacity of the list, and toArray(), which returns an array containing the list elements.

- **Association List:** The association list data structure discussed in Chapter 6.
- **Binary Search Tree:** A binary search tree implementing a set, with tree operations verified to preserve tree shape, ordering, and changes to tree contents. Supported operations include the ability to obtain and/or remove the maximum or minimum element in the tree in O(log n) time.
- **Circular List:** A circular doubly-linked list implementing a set interface.
- **Cursor List:** A list with a cursor that can be used to iterate over the elements in the list and, optionally, remove elements during the iteration. Method contracts include changes to the list contents and to the position of the iterator. Verified properties include that each complete pass over the list visits each element exactly once.
- **Hashtable:** The hash table data structure discussed in Chapter 7.
- **Priority Queue:** The priority queue data structure discussed in Chapter 5.
- **Singly-Linked List:** A null-terminated singly-linked list implementing a set interface.

8.1.1 Statistics

Figure 8-1 presents the number of Java and Jahob statements in the verified data structures in both graphical and tabular form. Figure 8-2 contains a graph of the same data normalized to the number of Java statements in the data structures. The first column in the table of Figure 8-1 presents the number of Java statements in the data structure implementations. The next two columns present the number of Jahob specification statements. We distinguish between class and method specification statements. The former pertain to the entire class, while the latter pertain to a specific method. Class specification statements consist of specification variable declarations and definitions (vardefs), class invariants, and class modifiers, such as claimedBy annotations. Method specification statements consist of method contracts and annotations within the method body, not including proof commands. The last column presents the number of proof commands in the verified data structures.

In general, the number of class specification statements stays more or less constant regardless of the number of Java statements in the data structure. The number of method specification statements, however, tends to grow in rough proportion to the number of Java statements. This is because the size of the implementation, as measured by the number of Java statements, often correlates with the number of



Data Structure	Java Statements	Class Specifications	Method Specifications	Proof Commands	Total
Array List	121 (51%)	14 (6%)	75 (31%)	29 (12%)	239
Association List	68 (58%)	13 (11%)	36 (31%)	0 (0%)	117
Binary Search Tree	140 (46%)	14 (5%)	45 (15%)	106 (35%)	305
Circular List	57 (57%)	17 (17%)	23 (23%)	3 (3%)	100
Cursor List	50 (42%)	34 (29%)	28 (24%)	7 (6%)	119
Hashtable	92 (27%)	24 (7%)	60 (17%)	168 (49%)	344
Priority Queue	85 (24%)	20 (6%)	41 (12%)	202 (58%)	348
Singly Linked List	38 (54%)	13 (19%)	19 (27%)	0 (0%)	70

Figure 8-1: Comparison of Java Statements, Specifications, and Proof Commands

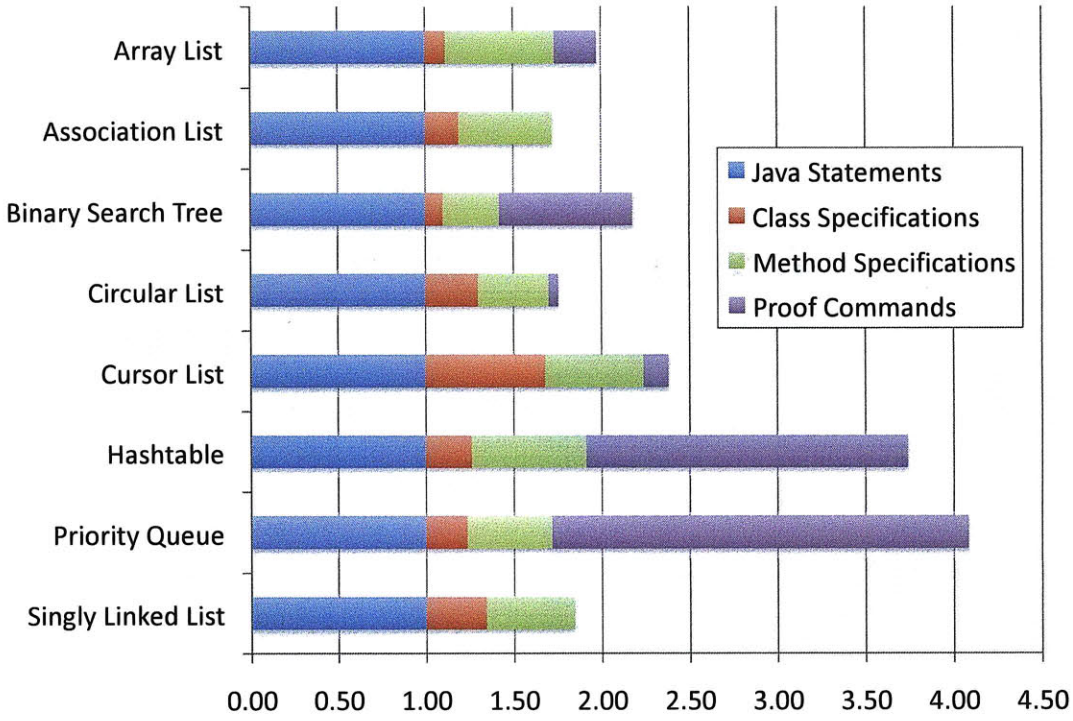


Figure 8-2: Normalized Comparison of Java Statements, Specifications, and Proof Commands

supported operations, as measured by the number of methods in the implementation. Class specification statements correspond to the description to the abstract state, which does not typically grow with the number of supported operations. But each method is associated with a method contract consisting of method specification statements, so the number of method specification statements tends to increase with the number of Java statements in the implementation.

The number of proof commands is more strongly correlated with the complexity of the data structure properties than the lines of Java or specification statements in the data structure. The data structures range from those that require little to no guidance in the form of proof commands, to those that use proof commands extensively. The latter, which include the binary search tree, hash table, and priority queue, represent the most difficult of the data structures to verify. For the binary search tree, this difficulty is due to the need to coordinate the efforts of multiple provers. The verification conditions for the binary search tree contain properties that depend on both shape and ordering properties, which are proved by different provers. The proof commands are therefore needed to identify intermediate lemmas encapsulating either shape or ordering properties for verification by the appropriate provers. The hash table data structure contains multiple levels of pointers that may be aliased. This complexity is reflected number of proof commands needed to verify the data structure. For the priority queue, the difficulty of the verification stems partly from multiple levels of indirection, and partly from the complexity of the implicit tree

structure. As a result of this difficulty, a substantial number of proof commands are needed for a successful verification.

Class Specification Statements

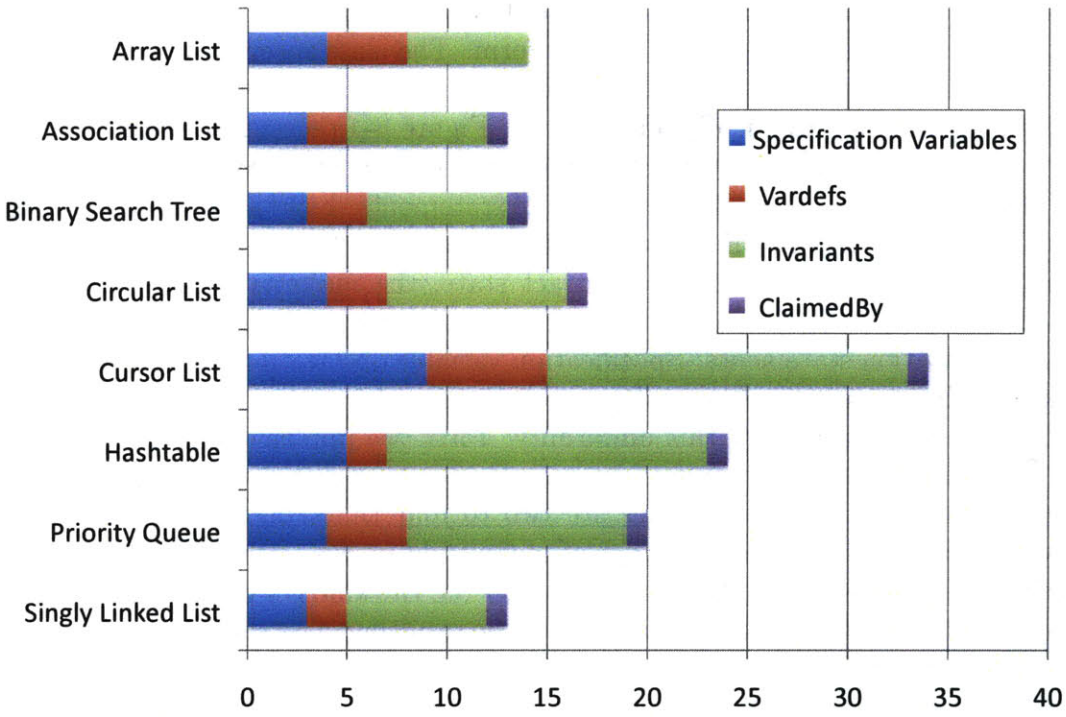
Figure 8-3 presents the breakdown of the different types of Jahob class specification statements in both graphical and tabular form. The class specification statements for the data structures consist of specification variable declarations, vardefs statements, class invariants, and claimedBy annotations. Figure 8-4 presents a graph of the same data, but as percentages of the total number of class specifications statements.

The distribution of the different types of statements shows a surprising uniformity. In spite of differences in the size of the data structure implementations, type of data structure, and complexity of the abstract state, the majority of class specification statements consist of class invariants. They make up between 40% and 60% of the class specification statements. Specification variable declarations and definitions each make up about 10–20%. ClaimedBy annotations make up less than 10%. There are several possible reasons for this uniformity. First, we implemented and specified all the data structures, so the uniformity may be due to stylistic similarities. Also, even though the largest data structures are two to four times bigger than the smallest, they are still roughly the same order of magnitude in size. And because they are all specified with respect to a set or map abstraction, the kinds of properties that need to be specified may be sufficiently similar that the resulting class specification statements have this uniformity of distribution. The complexity of the data structures is not necessarily well represented by these numbers, as complex data structure properties can be encoded in as few class invariants as simpler properties.

Method Specification Statements

Figure 8-5 presents the breakdown of the different types of Jahob method specification statements in both graphical and tabular form. Figure 8-6 contains a graph of the same data, but as percentages of the total number of method specification statements. Each method's contract typically contains requires, modifies, and ensures clauses, although some requires and modifies clauses are empty and therefore omitted from the specification. The remaining types of method specification statements consist of the declarations of local specification variables, loop invariants, specification assignment statements, non-deterministic assignment (havoc) statements, and the hidden modifier for the Java new statement.

The breakdown shows that method contracts, consisting of requires, ensures, and modifies clauses, make up more than half, and in some cases, over 80% of the method specification statements in a data structure. Since the number of methods tends to grow in proportion to the number of Java statements in the implementation, it is therefore not surprising that the number of method specification statements grow in rough proportion to the number of Java statements, as shown in Figure 8-1. After method contracts, specification assignment statements make up the largest portion of the remaining method specifications statements. These statements update the



Data Structure	Specification Variables	vardefs	Invariants	claimedBy	Total
Array List	4 (29%)	4 (29%)	6 (43%)	0 (0%)	14
Association List	3 (23%)	2 (15%)	7 (54%)	1 (8%)	13
Binary Search Tree	3 (21%)	3 (21%)	7 (50%)	1 (7%)	14
Circular List	4 (24%)	3 (18%)	9 (53%)	1 (6%)	17
Cursor List	9 (26%)	6 (18%)	18 (53%)	1 (3%)	34
Hashtable	5 (21%)	2 (8%)	16 (67%)	1 (4%)	24
Priority Queue	4 (20%)	4 (20%)	11 (55%)	1 (5%)	20
Singly Linked List	3 (23%)	2 (15%)	7 (54%)	1 (8%)	13

Figure 8-3: Breakdown of Class Specification Statements

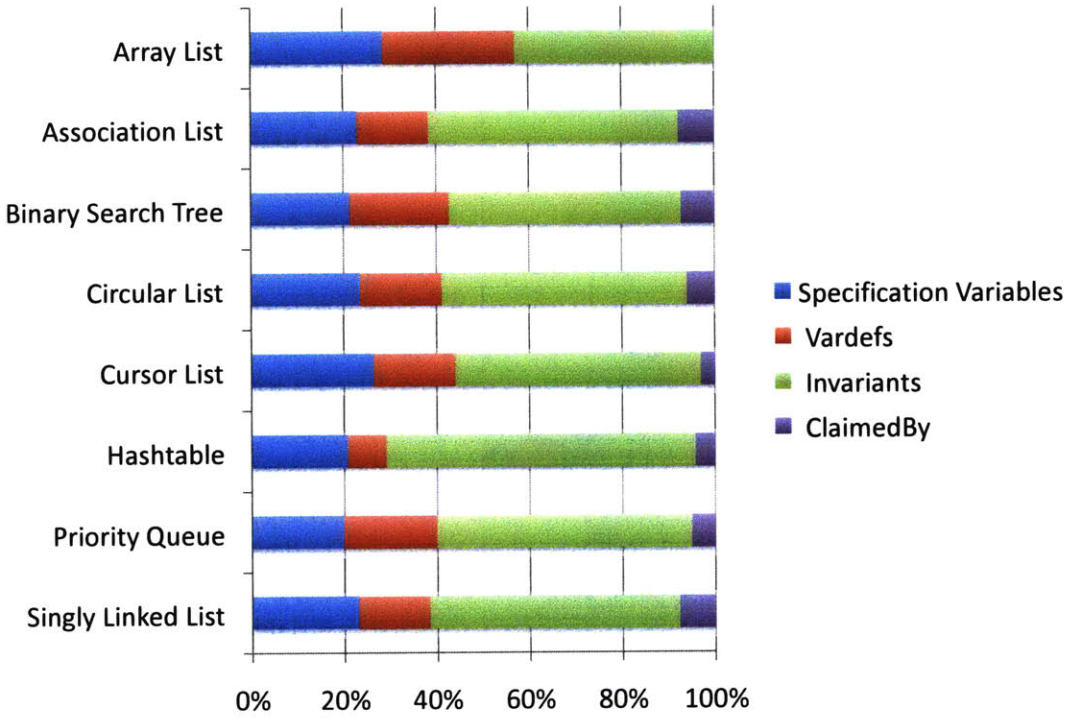


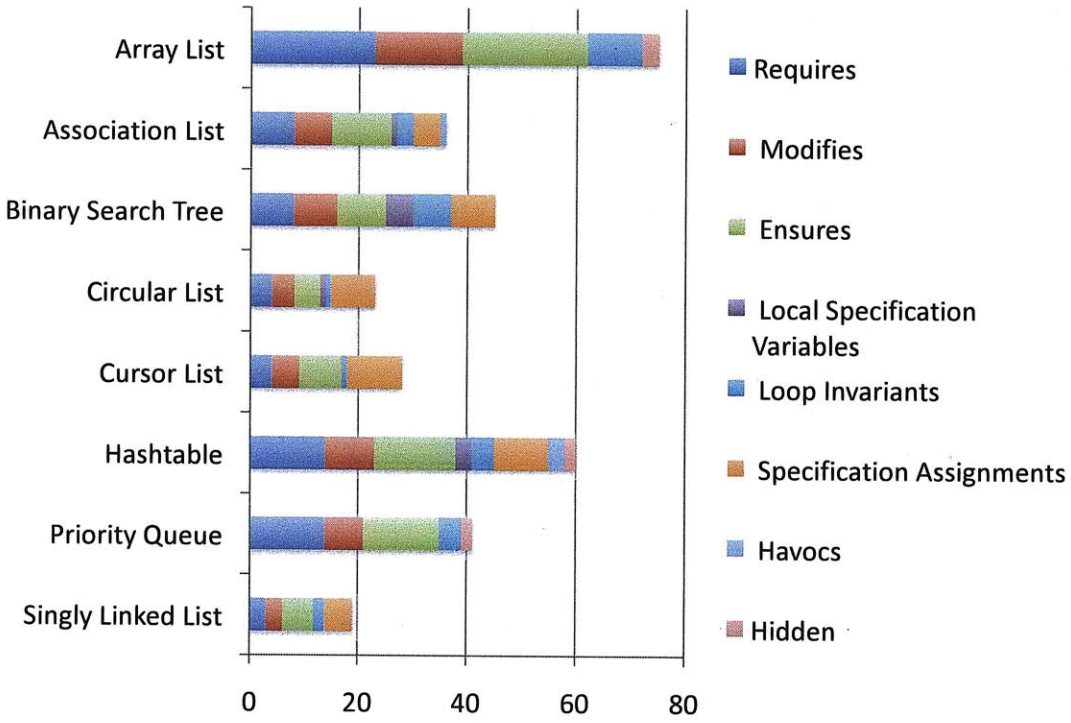
Figure 8-4: Breakdown of Class Specification Statements as Percentages

state of ghost variables that model the abstract state of the data structure. Loop invariants make up no more than 16% of the method specification statements for each data structure.

Proof Commands

Figure 8-7 presents the counts for each of the various proof language commands in both graphical and tabular form. There is one column in the table for each type of proof command used in our data structures, except for note commands, for which there are two columns. The first two columns contain entries for the number of note commands in the data structure implementations with and without a from clause, respectively. A from clause identifies a set of named facts for the provers to use when proving the new fact in the note command. Because the typical motivation for including a from clause is to limit the size of the assumption base so that the provers can prove the new fact in a reasonable amount of time, these numbers provide some indication of how sensitive the provers are to the size of the assumption base in each data structure. Figure 8-8 presents the same data, but as a percentage of the total number of proof commands in the data structure.

In general, the data structures use note commands (both with and without a from clause) much more extensively than any other type of proof language command. This fact reflects the strength of the underlying provers—it is often possible to guide the provers to an effective proof by either providing the intermediate lemmas that



Data Structure	Requires	Modifies	Ensures	Local Specification Variables
Array List	23 (31%)	16 (21%)	23 (31%)	0 (0%)
Association List	8 (22%)	7 (19%)	11 (31%)	1 (3%)
Binary Search Tree	8 (18%)	8 (18%)	9 (20%)	5 (11%)
Circular List	4 (17%)	4 (17%)	5 (22%)	1 (4%)
Cursor List	4 (14%)	5 (18%)	8 (29%)	0 (0%)
Hashtable	14 (23%)	9 (15%)	15 (25%)	3 (5%)
Priority Queue	14 (34%)	7 (17%)	14 (34%)	0 (0%)
Singly Linked List	3 (16%)	3 (16%)	6 (32%)	0 (0%)

Data Structure	Loop Invariants	Specification Assignments	havoc	hidden	Total
Array List	10 (13%)	0 (0%)	0 (0%)	3 (4%)	75
Association List	3 (8%)	5 (14%)	1 (3%)	0 (0%)	36
Binary Search Tree	7 (16%)	8 (18%)	0 (0%)	0 (0%)	45
Circular List	1 (4%)	8 (35%)	0 (0%)	0 (0%)	23
Cursor List	1 (4%)	10 (36%)	0 (0%)	0 (0%)	28
Hashtable	4 (7%)	10 (17%)	3 (5%)	2 (3%)	60
Priority Queue	4 (10%)	0 (0%)	0 (0%)	2 (5%)	41
Singly Linked List	2 (11%)	5 (26%)	0 (0%)	0 (0%)	19

Figure 8-5: Breakdown of Method Specification Statements

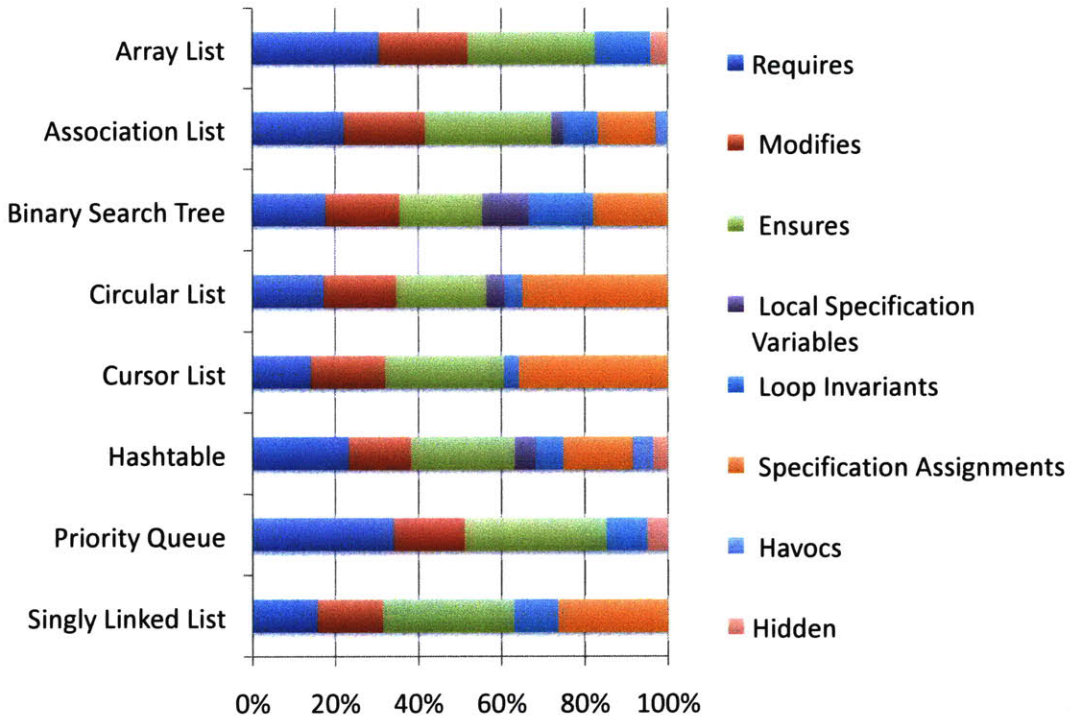
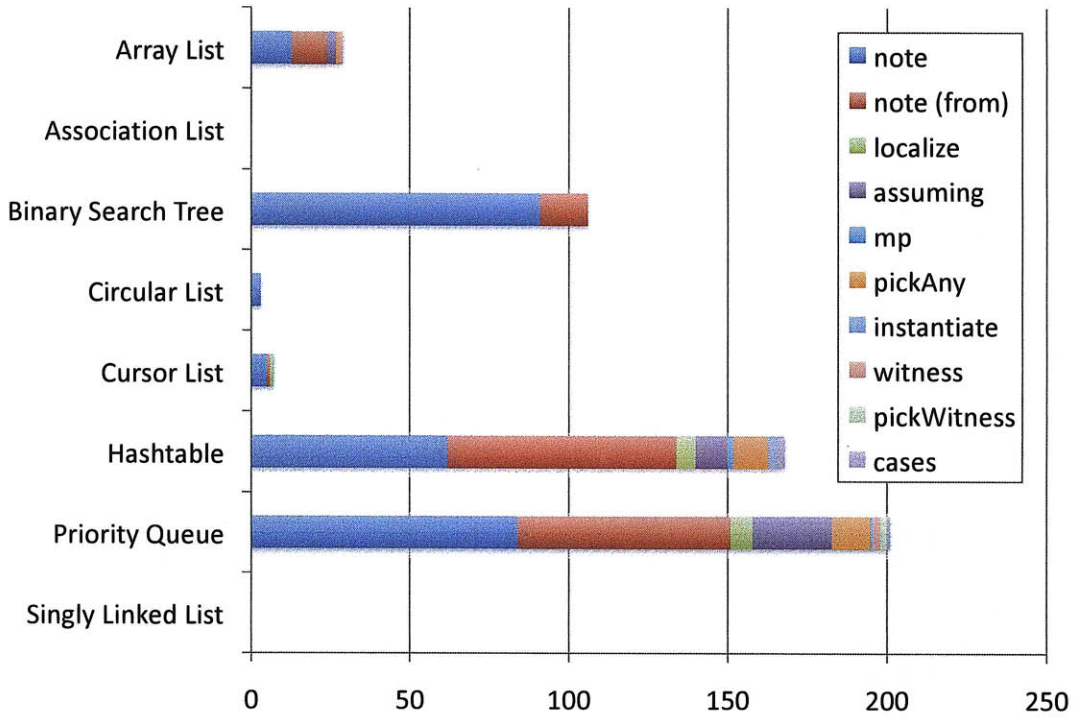


Figure 8-6: Breakdown of Method Specification Statements as Percentages

effectively guide the proof decomposition or by appropriately limiting the assumption base. In many cases, this type of guidance is all that is needed to direct the provers to a successful proof. For some data structures, it was also necessary to direct the system using other types proof commands. This necessity highlights the need for a proof language that supports not only high-level guidance, but also detailed proof steps.

Methods that modify the concrete data structure representation (such as methods that add or remove elements) tend to require at least some guidance in the form of proof commands. We attribute this property to the fact that the modifications often temporarily violate, then restore, key data structure invariants within the updated region of the data structure. In the absence of developer guidance, the provers must somehow determine where the invariants continue to hold, where they are violated, and what properties in the violated region enable the restoration of the invariants at the end of the data structure update. In our experience, provers often have difficulty with these tasks.

Methods that simply read data structure state tend to pose fewer difficulties. In more complex data structures, however, methods that access the concrete state in a non-trivial way still require some developer guidance to verify. With several exceptions, methods that only invoke other methods verify without guidance. This fact suggests that data structure clients, which only invoke data structure methods and do not access the concrete state of the data structure directly, should require substantially less developer guidance than data structure implementations.



Data Structure	note Commands	note (from) Commands	localize Commands	assuming Commands	mp Commands	pickAny Commands
Array List	13 (45%)	11 (38%)	0 (0%)	3 (10%)	0 (0%)	1 (3%)
Association List	0	0	0	0	0	0
Binary Search Tree	91 (86%)	15 (14%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Circular List	3 (100%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Cursor List	5 (71%)	1 (14%)	1 (14%)	0 (0%)	0 (0%)	0 (0%)
Hashtable	62 (37%)	72 (43%)	6 (4%)	10 (6%)	2 (1%)	11 (7%)
Priority Queue	84 (42%)	67 (33%)	7 (3%)	25 (12%)	0 (0%)	12 (6%)
Singly Linked List	0	0	0	0	0	0

Data Structure	instantiate Commands	witness Commands	pickWitness Commands	cases Commands	induct Commands	Total
Array List	0 (0%)	1 (3%)	0 (0%)	0 (0%)	0 (0%)	29
Association List	0	0	0	0	0	0
Binary Search Tree	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	106
Circular List	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	3
Cursor List	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	7
Hashtable	2 (1%)	0 (0%)	0 (0%)	3 (2%)	0 (0%)	168
Priority Queue	1 (0.5%)	2 (1%)	2 (1%)	1 (0.5%)	1 (0.5%)	202
Singly Linked List	0	0	0	0	0	0

Figure 8-7: Proof Command Counts for Verified Data Structures

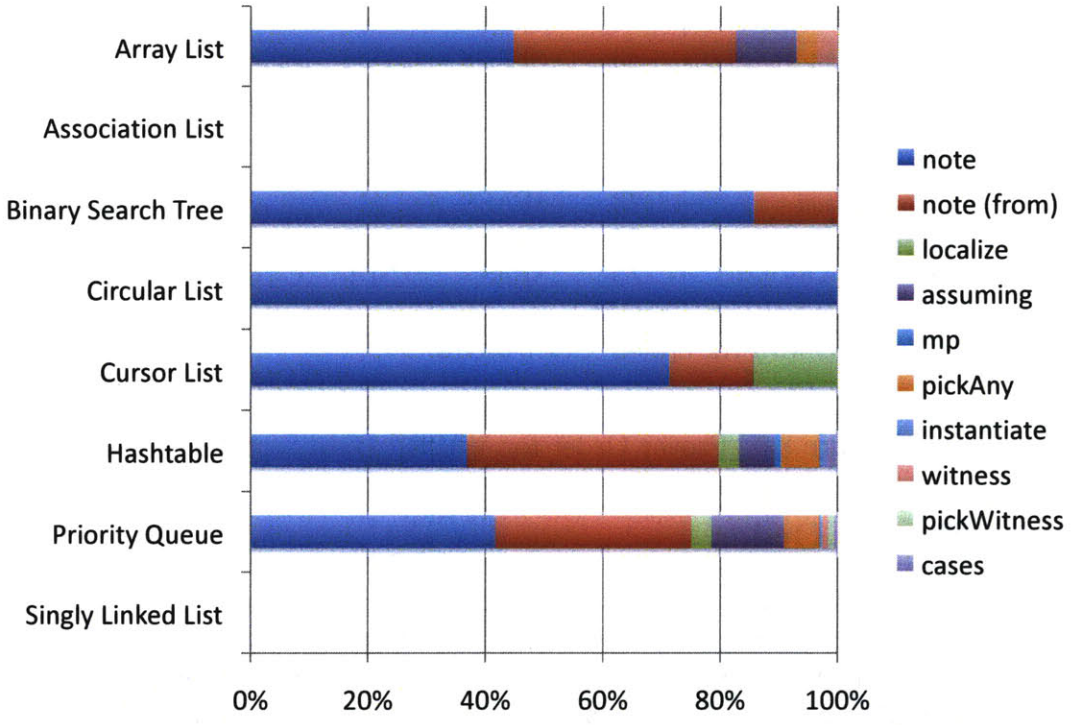


Figure 8-8: Proof Command Counts for Verified Data Structures (as Percentages)

8.2 Provers

All of the data structures were verified using a combination of automated theorem provers, decision procedures, and proof commands written in Jahob's integrated proof language, without the need for external interactive proofs. Specifically, the provers used in the verification consist of:

- Jahob's internal syntactic prover
- the MONA decision procedure [67] (for verifying shape properties [159]),
- the first-order prover SPASS [156],
- Jahob's internal cardinality prover,
- the SMT provers CVC3 [60] and Z3 [116, 117], and
- Isabelle [122] (through an automated, general-purpose theorem proving tactic).

These provers are described in more detail in Chapter 3. In previously published results [163], we also used the first-order prover E [149] and proofs written interactively using Isabelle.

Data Structures	Provers				
Array List	Z3	SPASS†	CVC3	Isabelle	
Association List	Z3	SPASS			
Binary Search Tree	MONA	SPASS	Z3		
Circular List	Z3	MONA			
Cursor List	Z3	SPASS			
Hash Table	Z3	SPASS†	Isabelle		
Priority Queue	Z3	SPASS	Card	CVC3	Isabelle
Singly Linked List	SPASS				

† Invoked with the option :OrderAxioms

Figure 8-9: Prover Order for Verified Data Structures

8.2.1 Prover Order

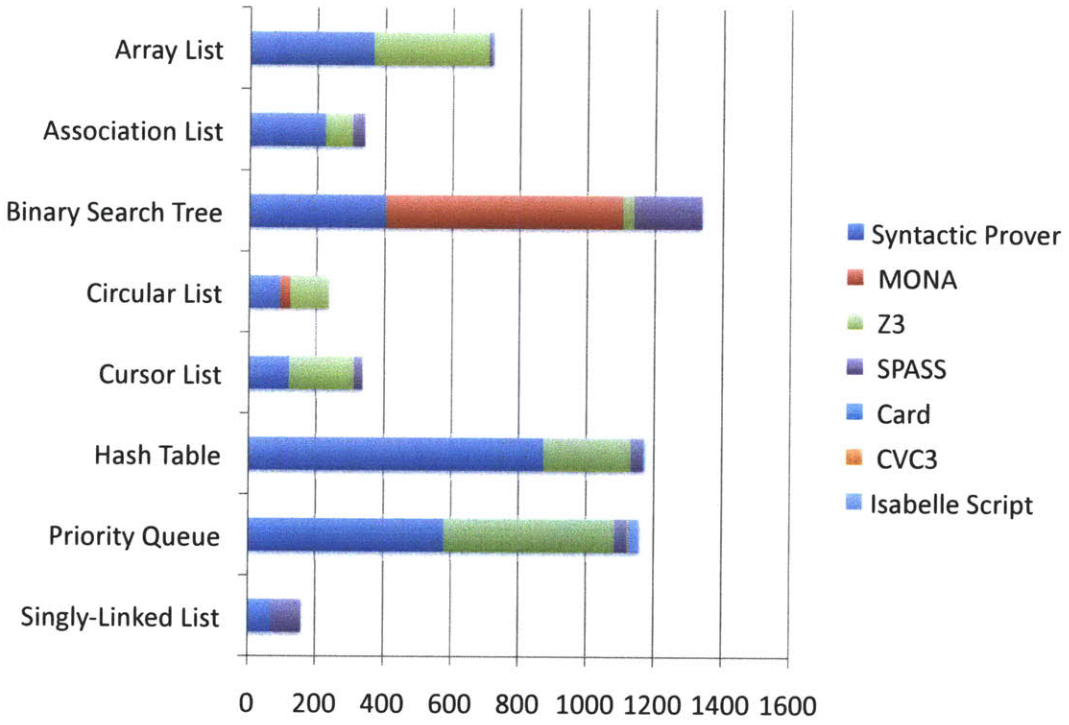
Figure 8-9 gives the order in which we applied the theorem provers and decision procedures for each data structure, and the prover-specific options that we used, if any. It does not include the internal syntactic prover, which is always the first prover that Jahob applies to a given formula. This occurs as part of the formula splitting process.

For most of the data structures, we use Z3 as the first prover (after the syntactic prover), because it is able to prove many sequents quickly. The only data structures for which we apply a different prover first are the binary search tree and singly linked list data structures. The binary search tree generates many verification conditions involving shape properties. We use MONA as the first prover because it is the only one able to handle such properties. The circular list, which also uses MONA, does not generate as many verification conditions involving shape properties; we use MONA as the second prover, after Z3, because the verification completes faster this way. For many of the data structures, SPASS is the second prover in the sequence. Although it is not as fast as Z3 on many of the sequents, it is able to prove some sequents that Z3 is unable to prove. We use SPASS as the only prover for the singly linked list data structure, because it can prove all the sequents generated, with the help of only the syntactic prover. (Z3 is able to prove some of the sequents for the singly linked list faster than SPASS, but there are some sequents it is not able to prove.)

The only prover-specific option used in the verification is the OrderAxioms option for the first-order prover SPASS. It directs Jahob to add to the assumptions given to SPASS a set of axioms for arithmetic ordering. We have found that this axiomatization, while incomplete, enabled SPASS to effectively reason about arithmetic properties in the verification of the array list and hash table data structures.

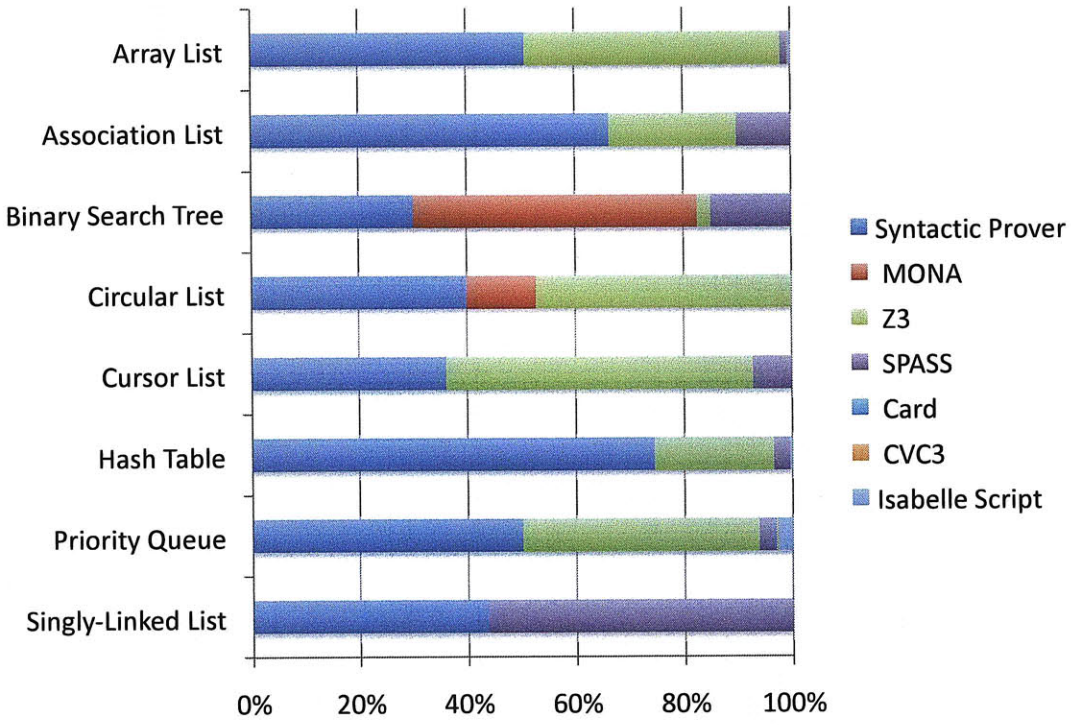
8.2.2 Sequents Proved

Figure 8-10 presents the number of sequents proved by each theorem prover or decision procedure for each verified data structure in both graphical and tabular form. A blank entry in the table indicates that the corresponding theorem prover or decision



Data Structure	Syntactic Prover	MONA	Z3	SPASS	Card	CVC3	Isabelle Script	Total
Array List	367		341	9		1	3	721
Association List	226		80	34				340
Binary Search Tree	405	701	34	199				1339
Circular List	95	30	112					237
Cursor List	122		191	24				337
Hash Table	875		259	35			4	1173
Priority Queue	581		505	35	3	3	30	1157
Singly Linked List	70			90				160

Figure 8-10: Number of Proved Sequents for Verified Data Structures



Data Structure	Syntactic Prover	MONA	Z3	SPASS	Card	CVC3	Isabelle Script
Array List	51%		47%	1%		0.1%	0.4%
Association List	66%		24%	10%			
Binary Search Tree	30%	52%	3%	15%			
Circular List	40%	13%	47%				
Cursor List	36%		57%	7%			
Hashtable	75%		22%	3%			0.3%
Priority Queue	50%		44%	3%	0.3%	0.3%	3%
Singly Linked List	44%			56%			

Figure 8-11: Proved Sequents by Prover (as Percentages)

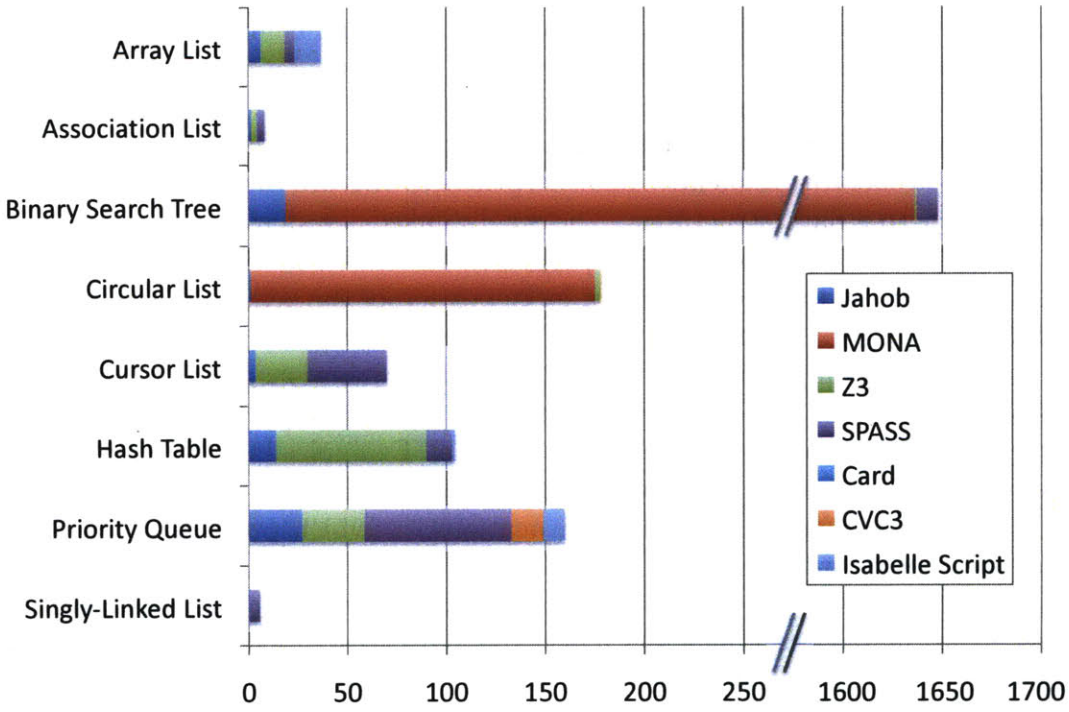
procedure was not used during the verification. The final column presents the total number of sequents proved. Figure 8-11 presents the same data as percentages of the total number of proved sequents.

For many of the data structures, the syntactic prover, together with Z3, discharges nearly all the generated sequents. The exceptions to this are the singly-linked list and binary search tree data structures. The singly-linked list does not use Z3. The binary search tree involves many shape properties, which we handle using MONA. The syntactic prover, by itself, discharges a large portion of the generated sequents for all the data structures. In many cases, this is due to formulas in which the consequent is in the set of assumptions, modulo splitting of conjunctions, and simple syntactic transformations. This can occur when properties in method preconditions correspond to proof obligations for checks within the method. Examples include null dereference checks and checks of method preconditions for method invocations. It can also occur when there are sequences of method invocations, in which postconditions of called methods earlier in the sequence fulfill preconditions of methods later in the sequence. The system also uses the syntactic prover heavily in conjunction with proof commands, as in the priority queue and hash table data structures. The syntactic prover handles the case where note commands label facts that are already in the assumption base. It also handles the case where a lemma proved using a proof command corresponds to a sequent from the generated verification conditions that needs to be discharged.

8.2.3 Verification Times

Figure 8-12 presents, for each theorem prover or decision procedure, the time it spent trying to proving the sequents it attempted to prove (top table), and the amount of that time spent on unsuccessful proof attempts for sequents that were later proved by other provers (bottom table). The difference between those two times is the time spent on successful proofs. For example, the Z3 entries for Association List are 4.5 and (1.8). This indicates that, for the association list data structure, the Z3 theorem prover spent 4.5 seconds trying to prove sequents, of which 1.8 seconds were spent on unsuccessful proof attempts for sequents that were later proved by SPASS. The remaining 2.7 seconds were therefore spent on successful proofs. The final column in the top table presents the total verification time, which includes the time spent in the verification condition generator, splitter, syntactic prover, and any applied decision procedures or theorem provers. The first column in the same table presents the time spent in Jahob. The values in this column represent an upper bound on the time spent in the syntactic prover, which is integrated into the splitter. We computed these values by subtracting the time spent in the decision procedures and theorem provers from the total time. Figure 8-12 also includes a graph of the time the provers spent trying to prove the sequents they attempted to prove (top table). Figure 8-13 presents the same data, but as percentages of the total verification time for each data structure.

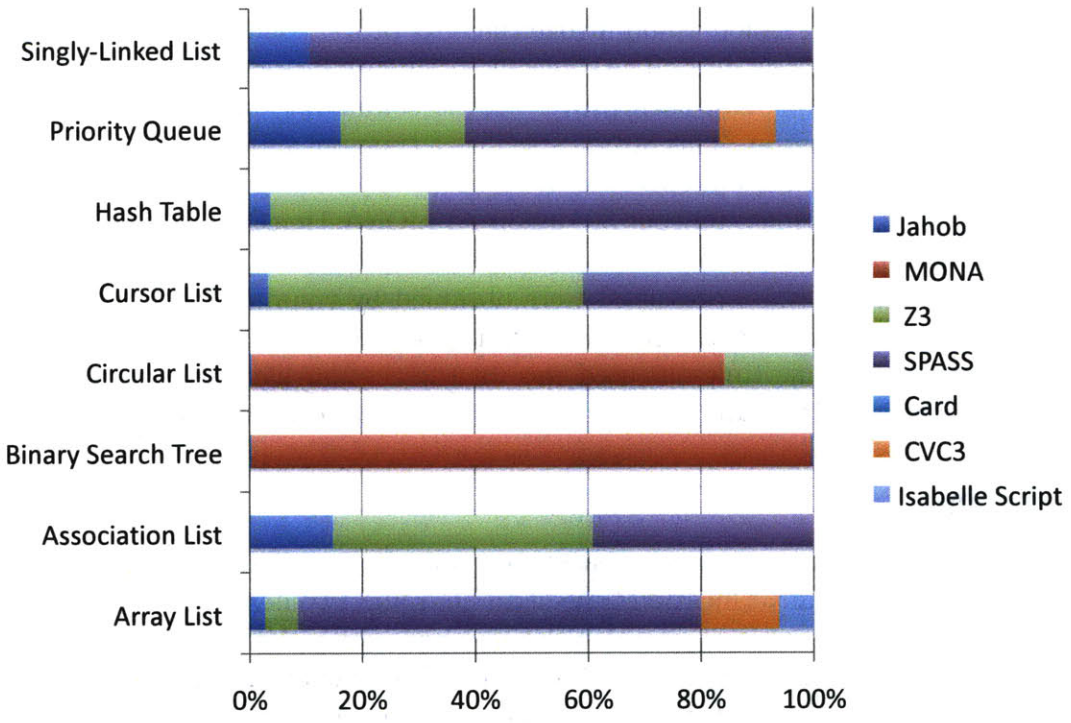
We obtained the results presented in these figures by invoking the provers as a sequential cascade, using time-outs. If one prover failed to prove a given sequent within



Data Structure	Jahob	MONA	Z3	SPASS	Card	CVC3	Isabelle Script	Total
Array List	6.1		13.1	155.9		30.3	13.2	218.6
Association List	1.5		4.5	3.9				9.9
Binary Search Tree	19.0	3641.9	1.1	14.0				3676.0
Circular List	1.1	174.0	32.7					207.8
Cursor List	3.4		54.5	39.8				97.8
Hash Table	13.8		99.0	238.5			1.3	352.7
Priority Queue	27.3		36.4	74.5	0.0010	16.5	10.8	165.4
Singly-Linked List	0.6			4.9				5.5

Data Structure	Jahob	MONA	Z3	SPASS	Card	CVC3	Isabelle Script	Total
Array List	-		(0.5)	(151.0)		(30.3)	(0.0)	181.8
Association List	-		(1.8)	(0.0)				1.8
Binary Search Tree	-	(2024.7)	(0.0)	(3.5)				2028.2
Circular List	-	(0.0)	(29.7)					29.7
Cursor List	-		(27.9)	(0.0)				27.9
Hash Table	-		(22.8)	(225.6)			(0.0)	248.4
Priority Queue	-		(5.0)	(0.3)	(0.0008)	(0.2)	(0.0)	5.5
Singly-Linked List	-			(0.0)				0.0

Figure 8-12: Verification Times (in seconds) for Verified Data Structures



Data Structure	Jahob	MONA	Z3	SPASS	Card	CVC3	Isabelle Script	Total
Array List	3%		6%	71%		14%	6%	100%
Association List	15%		46%	39%				100%
Binary Search Tree	1%	99%	0.03%	0.4%				100%
Circular List	1%	84%	16%					100%
Cursor List	4%		56%	41%				100%
Hashtable	4%		28%	68%			0.4%	100%
Priority Queue	16%		22%	45%	0.0006%	10%	7%	100%
Singly Linked List	11%			89%				100%

Data Structure	Jahob	MONA	Z3	SPASS	Card	CVC3	Isabelle Script	Total
Array List	-		0.2%	69%		14%	0%	84%
Association List	-		18%	0%				18%
Binary Search Tree	-	55%	0%	0.1%				55%
Circular List	-	0%	14%					14%
Cursor List	-		29%	0%				29%
Hashtable	-		6%	64%			0%	70%
Priority Queue	-		3%	0.2%	0.0005%	0.1%	0%	3%
Singly Linked List	-			0%				0%

Figure 8-13: Verification Times for Verified Data Structures (as Percentages)

the specified time-out, Jahob would invoke the next prover. Jahob provides two mechanisms for specifying time-outs. The -timeout command line option specifies a general time-out for all provers. But each prover on the command line can also be invoked with a :TimeOut=t option. This option specifies a time-out t for the given prover that overrides the general time-out specified by -timeout. For example, running Jahob with the command line options -timeout 30 -usedp Z3 MONA:TimeOut=60 would result in Z3 being run with a time-out of 30 seconds, and MONA being run with a time-out of 60 seconds. The results presented were obtained by running Jahob with -timeout 30 except for the binary search tree data structure, which was run with -timeout 50. The only prover-specific time-out used was for the circular list data structure, which was verified using a 60 second time-out for the MONA decision procedure.

Discussion

Most of the data structures verify within several minutes. The outlier is the binary search tree with a total verification time of an hour and two minutes, primarily due to the amount of time spent in the MONA decision procedure. MONA spends more than half of that time in unsuccessful proof attempts of sequents that are later proved by Z3.[1] MONA is also responsible for the majority of the verification time in the circular list. But in that case, it is the last prover in the cascade, so all of the time is spent on successful proofs. SPASS is responsible for a significant portion of the verification time in the array list, cursor list, hash table, and priority queue data structures. For the array list and hash table, SPASS spends much of that time on unsuccessful proof attempts. But for the cursor list and priority queue, SPASS spends it primarily on successful proofs. SPASS is only responsible for a small fraction of the verification time in the other four data structures that use it.

In general, the sequential verification times presented here represent upper bounds on the time that it would take to verify the data structures in parallel. A prover that is unable to prove a formula will often time-out, causing the proof of that formula to require at least the amount of time specified by the time-out, plus the time required for the successful proof. In the meantime, no other provers can proceed. (Consequently, the system is more efficient when provers that eventually fail to prove a formula, do so quickly.) In parallel mode, Jahob creates multiple processes, each of which is responsible for proving a single sequent. Provers are still executed as a cascade on a given sequent. But the effect of unsuccessful proof attempts on the overall verification time is generally significantly reduced, as provers can proceed with other sequents simultaneously.

Prover Efficiency

Consider the efficiency of a prover as defined by the number of sequents proved per unit time. Although the syntactic prover is responsible for proving many of the sequents in the verification, it does so using a small fraction of the total verification

[1] We use Jahob in sequential mode to measure the time spent in each prover. In practice, we can invoke the provers in parallel to obtain better performance on processors with multiple cores.

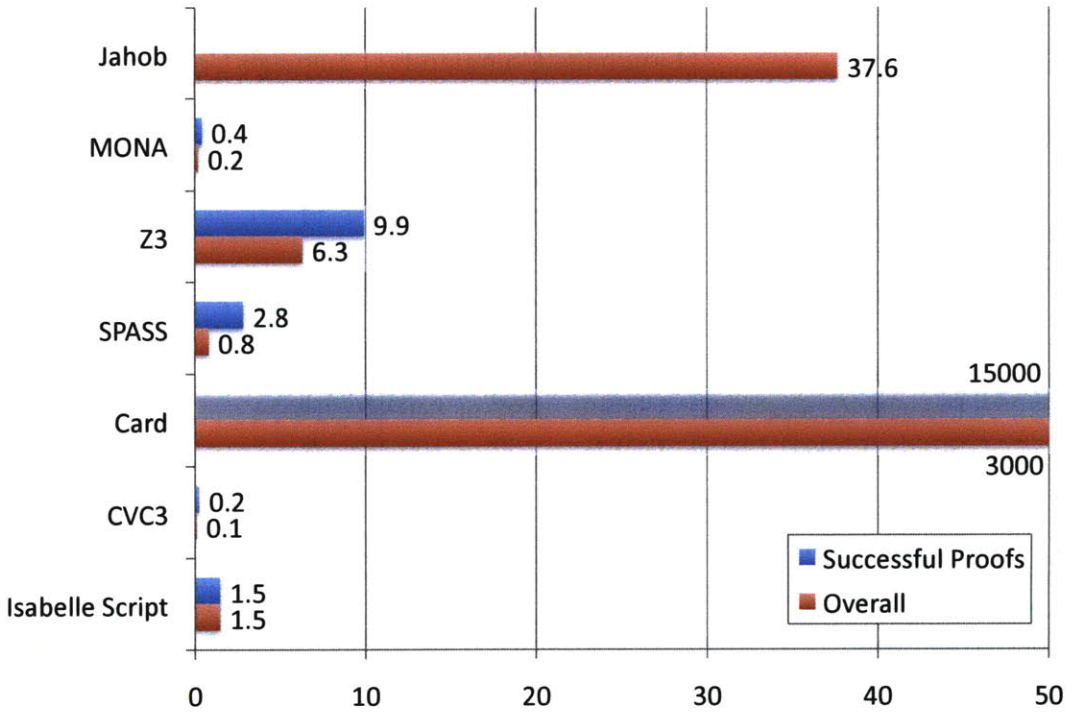


Figure 8-14: Prover Efficiency (in sequents/s)

time. The same is true for Z3. In contrast, the time spent in SPASS for the array list, cursor list, and hash table data structures is disproportionately large compared to the number of sequents proved. This disparity in efficiency is partly due to the inherent efficiency of the provers, and partly due to the ability of the provers to fail quickly on sequents they are not able to prove. The syntactic prover and Z3 both tend to fail quickly on sequents they cannot prove, while SPASS will sometimes execute until the time-out expires, resulting in a longer total time spent, even though this time is spent on proofs that do not eventually succeed. However, SPASS is also able to successfully prove difficult sequents that earlier provers in the cascade fail to prove. Similarly, the MONA decision procedure takes up much of the verification time in comparison to the number of sequents proved, both because it spends a lot of time on unsuccessful proof attempts, and because of the difficulty of the sequents proved. But it is able to successfully prove sequents involving shape properties that other provers are unable to prove. In short, the efficiency of a prover is affected by a combination of its inherent efficiency on successful proofs, its ability to fail quickly on sequents it is not able to prove, and the difficulty of the sequents it is proving.

Figure 8-14 presents the efficiency of the provers over all the verified data structures in sequents per seconds. We measure both efficiency on successful proofs and the overall efficiency, which takes into account the time spent on unsuccessful proof attempts. We measure efficiency on successful proofs by taking the total number of sequents proved by each prover over all the verified data structures, and dividing by the total time spent by that prover on successful proofs. For the overall efficiency, we

Data Structure	Without Proof Language Commands: Methods Verified	Without Proof Language Commands: Sequents Verified	With Proof Language Commands: Methods Verified	With Proof Language Commands: Sequents Verified
Array List	20 of 23	684 of 687	23	721
Association List	11 of 11	340 of 340	11	340
Binary Search Tree	0 of 9	743 of 884	9	1339
Circular List	2 of 5	215 of 229	5	237
Cursor List	7 of 8	330 of 331	8	337
Hash Table	6 of 15	968 of 999	15	1173
Priority Queue	5 of 14	797 of 840	14	1157
Singly Linked List	6 of 6	160 of 160	6	160

Figure 8-15: Effect of Proof Language Commands on Verification

divide by the total time spent by that prover, including time spent on unsuccessful proof attempts. For the syntactic prover, we compute only the overall efficiency, using the total time spent in Jahob. We were unable to measure the time spent in the syntactic prover separately because the syntactic prover is integrated into the formula splitting process. The overall efficiency for the syntactic prover is therefore a lower bound on its actual efficiency.

Not surprisingly, the simplest provers—the syntactic prover and the cardinality prover—were able to prove the most sequents per unit time. Among the remaining provers, Z3 is the most efficient. MONA was able to prove less than one sequent per second, but was able to handle shape properties not provable by the other provers. Because we used the Isabelle proof script as the last prover in the cascade for all the data structures for which it was used, our results did not include any measurements for time spent on failed proof attempts for Isabelle. Its overall efficiency is therefore the same as its efficiency on successful proofs.

8.3 Effect of Proof Language Commands

We next discuss the effect of Jahob's proof language commands on the verification. Jahob's proof language commands were necessary for the successful verification of all but two of our data structures. The number of proof commands used spanned a wide range, from a handful of proof commands for some data structures, to the extensive use of proof commands for others. We used ten different types of proof commands in the verification. We used the note command (both with and without a from clause) more extensively than any other proof command. However, other proof commands were also necessary for the successful verification of our data structures.

Figure 8-15 summarizes the effect that the proof language commands have on the verification. The first two columns of the table present the number of methods and sequents verified without proof language commands. We obtained these numbers by

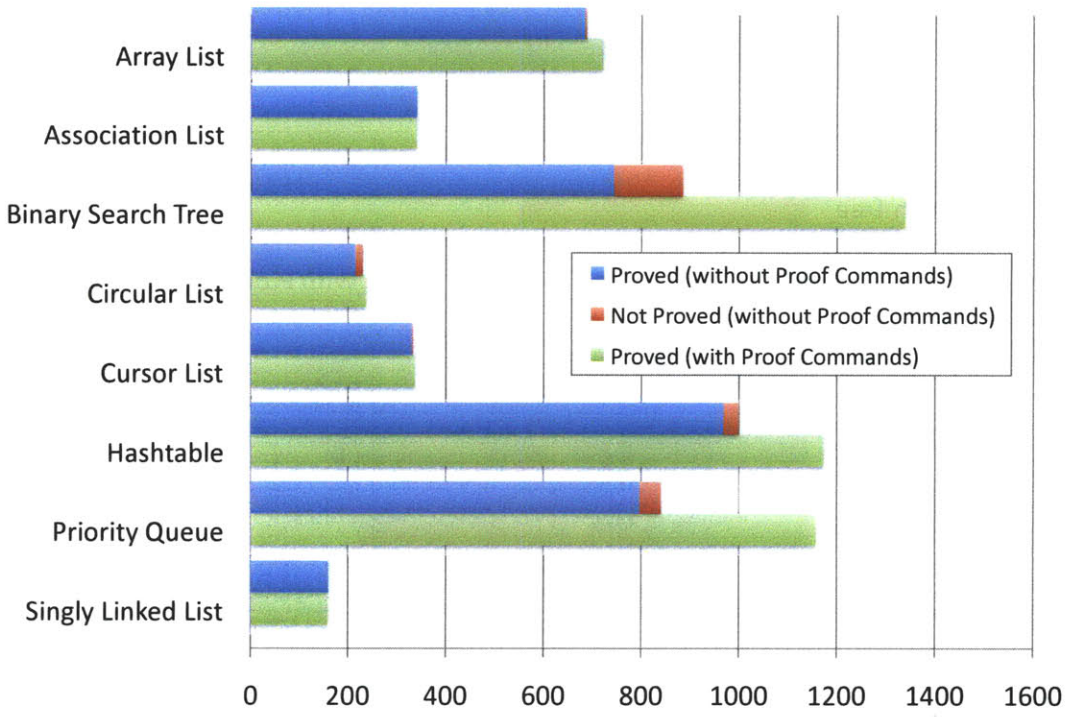


Figure 8-16: Sequents Proved with and without Proof Commands

removing all proof commands from the program, then attempting to verify the data structure. In general, the more complex the data structure, the more guidance the provers need to verify the data structure.

The final column of the table presents the total number of sequents required to fully verify the corresponding data structure implementations after adding the necessary proof language commands. Note that the number of sequents increases, in some cases significantly. This is because the proof commands force the provers to prove additional lemmas, which in turn correspond to additional sequents. The increase in the number of sequents reflects the difficulty of proving the complex sequents that failed to verify in the absence of developer guidance.

Figures 8-16 and 8-17 present the data from Figure 8-15 in graphical form. Figure 8-16 compares the number of sequents proved without and without proof commands for each data structure. It illustrates how the combined reasoning system is able to prove the large majority of the sequents without the use of proof commands. At the same time, it also illustrates how many intermediate lemmas are required to prove the small percentage of sequents that require the use of proof commands.

Figure 8-17 compares the number of methods proved with and without proof commands. In some data structures, such as the array list and cursor list, the proof commands needed are concentrated in a small number of methods with difficult proof obligations. In others, such as the cursor list, hash table, and priority queue, the proof commands are divided amongst many methods. For these data structures, proof commands are needed in many of the methods that directly modify the concrete program

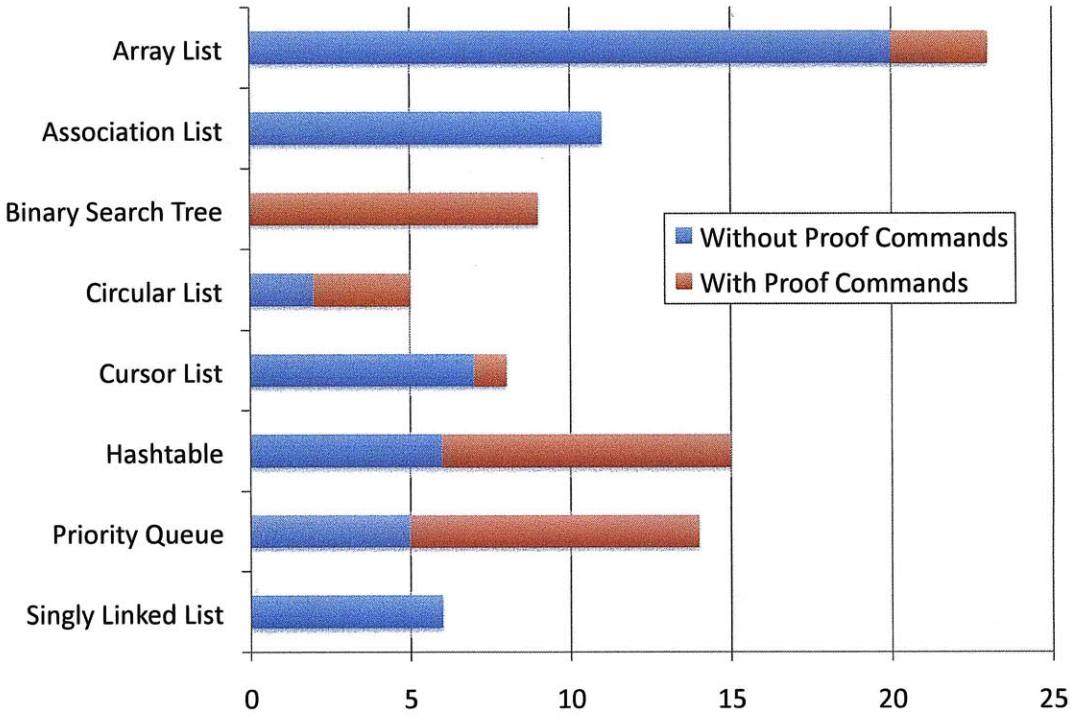


Figure 8-17: Methods Verified with and without Proof Commands

state. This is because of the complexity of the data structure properties that need to be proved. In the extreme case of the binary search tree, all the methods verified required proof commands. This is because the binary search tree properties combine shape properties and ordering properties, which are proved by different provers. The proof commands identify intermediate lemmas that separate out the relevant shape and ordering properties, so that the appropriate provers can prove them.

8.3.1 Sensitivity to Proof Language Commands

In the process of verifying a data structure, it is not uncommon to accidentally enter an incorrect proof command. In general, these errors are typically errors in the formulas given to the proof commands, and not errors in the type of proof command used. Proof commands asserting formulas that are false are usually quickly identified when the formula fails to prove. Formulas that are true, but not necessary to the proof, do not usually affect the ability of the provers to successfully verify the program. In theory, it is possible for many such formulas, if they were universally quantified, to blow up the size of the proof search space. However, we have not experienced this in practice. Due to the power of the automated provers integrated into Jahob, there are often many possible intermediate lemmas and proof commands that could be used to guide the combined reasoning system to a successful proof. Some lemmas and proof commands may lead to a shorter verification time than others. For the results presented here, we have not performed any systematic optimization of the

proof commands for verification time, since the verification time is generally small compared to the development time.

8.3.2 Discussion

We make the following observations:

- **Many Sequents:** The data structure correctness proofs involve a large number of sequents. It would be challenging and potentially impractical for a developer to use an interactive proof system to prove all of these sequents.

- **Prover Effectiveness:** The provers verify many methods and a large percentage of the sequents without developer guidance. This fact reflects the effectiveness of the provers in verifying even the complex verification conditions that arise during the verification of imperative linked data structures. Despite this effectiveness, however, the provers are capable of fully verifying only the simplest data structures.

Our results therefore support the use of a hybrid approach that uses developer guidance to leverage the strength of existing provers rather than relying solely on either interactive proofs or fully automatic verification. We have found Jahob's integrated proof language effective in supporting such an approach to enable the verification of the data structure implementations in our benchmark set.

8.4 Summary

In this chapter, we describe the empirical results from our verification of full functional correctness for a collection of imperative linked data structure implementations. These implementations include both array-based data structures, such as priority queues and hash tables, and recursive data structures, such as lists and trees. Our verified data structures implement set and map interfaces, which we specified in terms of the abstract state of the data structure. The verified invariants capture the internal correctness properties necessary for the correct operation of the data structure, while the verified method contracts completely capture the behavior of the corresponding method in terms of the data structure's abstract state.

Our empirical results support both our integrated reasoning approach and the use of our proof language. All of the data structures verified required the use of multiple provers. The combined reasoning system was able to handle properties beyond the reach of any single prover. The simplest provers were able to discharge many sequents efficiently. While more sophisticated provers were less time-efficient, they were able to handle sequents that other provers were unable to discharge.

The combined reasoning system was also able to prove the large majority of the generated sequents automatically. This result reflects the strength of the underlying provers. The remaining sequents, however, required the use of proof commands to verify. In general, these proof commands guided the proof of complex verification

conditions arising in methods that directly update the concrete state. The large number of sequents involved in these proofs makes it challenging and potentially impractical to use an interactive proof system to prove these sequents manually. Our proof language, however, enabled the effective proof of these sequents through the use of the integrated reasoning system.

Chapter 9

Related Work

Jahob builds on a large body of work in program verification, but is unique in its integration of both a proof language and the diversity of automated reasoning techniques supported. While many program verification systems focus the verification of partial correctness properties, this integration has enabled us to use Jahob to verify the full functional correctness of imperative linked data structure implementations. While our results are, in theory, obtainable using interactive theorem proving, the expertise and manual effort required for such an endeavor makes this approach difficult to use in practice. In equipping Jahob with a proof language, we have attempted to bridge the gap between program verification and interactive theorem proving, providing a level of automation and control that makes it feasible to tackle complex functional correctness properties in the presence of difficult verification issues such as aliasing and destructive updates. In contrast, techniques not based on theorem proving that have also been applied to the problem of data structure verification—such as shape analysis, type systems, and decidable logics—focus primarily on partial correctness, and are not able to address the full range of correctness properties that we have verified using Jahob.

In this chapter, we survey existing research in program verification, interactive theorem proving, and data structure verification, and discuss how the work presented in this thesis differs from other systems and the results obtained using them. We also briefly discuss verification techniques based on finitization, such as model checking, testing, and bounded verification. Systems based on these techniques are able to offer more automation, but are only able to verify the correctness of programs for finitely many executions. In contrast, Jahob and other program verification systems based on theorem proving are able to verify correctness for all possible executions.

9.1 Program Verification

Program verification systems based on assume-guarantee reasoning abound, and include Hob [92, 83, 91], ESC/Modula-3 [51], ESC/Java [56], ESC/Java2 [39, 34], Spec# [11, 12, 97], KeY [16], Krakatoa [55, 106], LOOP [22], Jive [110], and Jack [14, 32]. While many of these systems focus on the verification of partial correctness

properties, we have applied Jahob to the task of verifying the complex functional correctness properties that arise in imperative linked data structure implementations. Although the programs we have verified are small in comparison to those partially verified using other systems, we find that the results we have been able to obtain using Jahob successfully address difficult verification issues, such as aliasing and data abstraction, to an extent not explored in other systems.

9.1.1 Hob

The Hob system [92, 83, 91], Jahob's predecessor, supports verified data structure interfaces that use sets of objects to summarize the state of the data structure and the effect of data structure operations. Because the Hob specification language is based on sets, it is powerful enough to specify full functional correctness only for data structures that export a set interface. For data structures with richer interfaces (such as hash tables, lists, priority queues, and search trees) it can specify and verify some, but not all, correctness properties. Like Jahob, Hob integrates a variety of reasoning techniques to successfully discharge generated verification conditions, including the use of arbitrarily precise reasoning techniques within data structure implementations. But unlike Jahob, which can apply multiple reasoning techniques to a single formula, Hob lacks the ability to apply more than one technique within a Hob module. Hob also does not include support for a proof language.

9.1.2 Jahob

Jahob is also the subject of Viktor Kuncak's PhD thesis [82], which focuses on the techniques that support our integrated reasoning approach, and, in particular, the approximation techniques for using Jahob in conjunction with first-order logic resolution-based provers, field constraint analysis, and BAPA. At the time, we were able to use our system to verify non-instantiable data structures, though some of them were functional, while others did not preserve abstraction (i.e. adding a node to a linked list, as opposed to adding an object to the set that the list implemented). The system supported note commands with from clauses, but not the other commands of the proof language. Proofs that required more detailed guidance were written using external proof assistants.

This thesis focuses on the application of the Jahob system to the verification of full functional correctness for imperative linked data structures, and on the integrated proof language which makes this verification possible. In the time since the completion of Viktor's thesis, we have continued to improve Jahob's internal algorithms, interfaces to internal and external provers, and designed, implemented, and proved the soundness of our proof language. These improvements have enabled us to verify substantially more sophisticated data structures and specifications than we were able to verify using the earlier incarnation of the system. The proof language also enables us to avoid the use of external proofs in the verification. All the examples presented in this thesis are imperative data structures that preserve abstraction. With the exception of the two examples that use the MONA decision procedure, all of our verified

data structures are instantiable.

9.1.3 ESC/Modula-3 and ESC/Java

The family of extended static checking tools includes ESC/Modula-3 [51] and ESC/Java [56]. As their names imply, ESC/Modula-3 operates on Modula-3 programs, while ESC/Java operates on Java programs. Although ESC/Modula-3 and ESC/Java are designed to be bug finding tools, and not program verification systems, we include them here because of their similarity to Jahob in the way in which programs are specified, and in the way in which they generate and prove verification conditions. In contrast to Jahob, which is targeted towards proving complete functional interfaces for data structure implementations, ESC/Modula-3 and ESC/Java were designed for the purpose of identifying errors in components of large systems programs (e.g. the Modula-3 runtime system). One result of this design difference is that they are more fully-automated than Jahob in that they do not require loop invariants, but generate unsound approximations for loop invariants where needed. As a result, their approach is not sound. Because the goal is not to prove complex program correctness properties, but to identify bugs, they do not require the powerful reasoning technologies that Jahob supports. Both ESC/Modula-3 and ESC/Java support a single prover, Simplify, and do not provide a mechanism for the user to guide the prover when it encounters a difficult proof.

Like Jahob, both ESC/Modula-3 and ESC/Java support modular checking of method contracts, though ESC/Java does not check modifies clauses due to a lack of support for data abstraction. ESC/Modula-3 provides this support through *abstract variables*, which are analogous to Jahob's dependent specification variables. Instead, ESC/Java supports ghost fields and the use of *object invariants* to establish the necessary abstraction functions. This is analogous to Jahob's ghost variables and invariants, though the invariant enforcement policies differ. ESC/Modula-3 and ESC/Java have both been used to identify bugs in large programs with partial specifications, but have not been used to verify full functional correctness. Unlike Jahob, neither supports integrated reasoning or an integrated proof language.

9.1.4 ESC/Java2

ESC/Java2 [39, 34] is an ongoing project based on ESC/Java that adopts JML [94, 93] (Java Modeling Language) as its specification language. ESC/Java2 supports modular checking of method contracts, *model fields* (which are analogous to Jahob's dependent variables), and ghost fields (which are analogous to ghost variables). It currently interfaces to a single prover, Simplify, though work to incorporate multiple automated and interactive provers is in progress [77]. One difficulty that arises in incorporating external reasoning techniques into ESC/Java2 is the need to map the JML specification language into the appropriate logics, as work to clarify the semantics of JML is still ongoing. This difficulty highlights the benefit that Jahob derives from using HOL as a specification language—HOL has well-defined semantics and, using formula approximation, maps naturally to logic subsets supported by a variety of automated

provers, decision procedures, and proof assistants. ESC/Java2 has been applied to a partially specified electronic voting system [39] and to verify a number of safety and correctness properties of an implementation of SSH [129]. Unlike Jahob, ESC/Java2 does not support an integrated proof language, and has not been used to verify data structure implementations comparable to the ones presented in this thesis.

9.1.5 Spec#

The Spec# programming system [11, 12, 97] consists of the Spec# programming language, compiler, and the Boogie static program verifier [10]. Spec# takes a combined dynamic and static checking approach to program verification, with runtime checks inserted by the compiler and the Boogie verifier comprising the static component. The Spec# programming language is an extension of C# with contracts. Boogie operates over its own language, BoogiePL, a simple procedural language to which Spec# programs are translated after first being compiled into CIL (.NET's Common Intermediate Language).

Spec# supports modular checking of method contracts and object invariants, model fields, as well as an ownership system that determines when object invariants are required to hold. Contracts may include pure program expressions as well as universal and existential quantifiers, though Boogie is able to check only quantification over integers. Boogie includes a framework of abstract interpretation that is able to infer some loop invariants, though explicit loop invariants may also be required. The current default prover for Boogie is Z3, though the original prover, Simplify, is still supported.

Spec# supports an assert statement which is similar to Jahob's note command, but without from clauses. Boogie 2 additionally supports a call-forall statement, which, in conjunction with lemma procedures (similar to proof methods in Jahob), makes it possible to provide some guidance to the system in the verification of universally-quantified lemmas and implications [96]. This support, however, does not include the ability to control the assumption base, use induction, or apply the other first-order rules of deduction that are part of Jahob's proof language. Jahob also supports a greater number and diversity of automated reasoning systems. And because Boogie does not currently support checking of quantification over objects, which is necessary for many of the invariants we verified, we expect that Boogie would not be able to verify the data structures in our collection.

9.1.6 KeY

The KeY tool [16] is an interactive, deductive program verification tool that operates on Java Card programs annotated with JML or OCL (UML's Object Constraint Language) specifications. It also supports some Java features not included in Java Card. KeY compiles specifications into first-order logic, then generates verification conditions by symbolic execution of the source code, with induction for loops and recursion. These proof obligations are written in a dynamic logic for Java Card. They can be proved interactively using KeY, which also supports automated proof

search, Simplify, and SMT provers for increasing the automation of proofs. KeY supports modular verification of method contracts and invariants, as well as the use of model fields in specifications.

KeY has been used to interactively verify a Java Card implementation of Mondex, a protocol for electronic purses [148, 153]. The properties verified include functional properties as well as security properties, but not a verified abstraction. Specifically, the method postconditions do not preserve abstraction but state the effects of executing the method at a concrete level. KeY has also been used to verify a Java Card API reference implementation [115], but with the assumption that there is no inter-object data aliasing, one of the challenges addressed in the data structures verified using Jahob. Although KeY supports multiple provers, it does not include program analyses such as shape analysis. Unlike Jahob, which supports the direct embedding of proofs into programs through the integrated proof language, KeY uses a separate environment for proofs, with integration provided by means of a graphical user interface. While KeY has been used to prove the correctness of an insertion operation into a TreeMap [142], and can, in theory, be used to prove full functional correctness for more extensive examples, we are not aware of any results demonstrating its use on data structure implementations comparable to the ones we verified using Jahob.

9.1.7 Krakatoa

Krakatoa/Why [55, 106] are tools for the deductive verification of Java programs. Krakatoa interprets Java programs annotated with JML specifications; the related tool Caduceus interprets C programs annotated with specifications in a JML-like language. Krakatoa and Caduceus both generate programs that serve as input to the Why tool, a verification condition generator. These programs are written as functions annotated with pre- and postconditions written in a polymorphic first-order logic with built-in equality and arithmetic.

Like Jahob, Why produces verification conditions using weakest precondition semantics [54], and output to a number of different automated and interactive provers, including Simplify, Z3, CVC3, Isabelle, and Coq. Unlike Jahob, Why does not support an integrated proof language, or include program analyses such as shape analysis.

9.1.8 LOOP

The LOOP tool [22] verifies programs written in Java and annotated with either CCSL (Coalgebraic Class Specification Language) or JML specifications. The tool generates theories in higher order logic that can be proved interactively using either PVS or Isabelle. The LOOP tool has been used to verify an invariant of the Java Vector class [69], termination specifications for the Java Card API [130], as well as more complete functional specifications for Java Card's Application Identifier class [23], though the verified properties are not as difficult as those addressed in our verified data structures. For instance, the concrete state of the AID class is immutable, so issues of aliasing in the presence of destructive updates do not apply. Unlike Jahob,

it does not support integrated reasoning or an integrated proof language, but uses external interactive theorem provers to discharge the generated proof obligations.

9.1.9 Jive

The JIVE (Java Interactive Verification Environment) tool [110] was originally designed for the verification of SVENJA, a subset of sequential Java with object-oriented features, with specifications written in the ISL (interface specification language) ANJA, which is based on first-order logic. Verification conditions were generated in a Hoare-style programming logic and verified interactively using PVS. It has been used to verify a doubly-linked list of integers [88] with respect to a list interface.

JIVE is being reimplemented to verify specifications written in JML, with programs written in Diet Java Card (DJC). Work is ongoing to define a formal semantics for JML by translation into first-order logic [42] and into the underlying theories of a given proof assistant [43].

9.1.10 Jack

The Jack (Java Applet Correctness Kit) tool [14, 32] verifies Java and Java Card programs annotated with JML specifications, as well as Java bytecode programs annotated with BML (Bytecode Modeling Language) [33] specifications. Like Jahob, Jack generates verification conditions using weakest preconditions. In Jack, these verification conditions are first-order logic formulas, which can be proved using either Simplify or the Coq proof assistant. To aid interactive verification, Jack includes specialized Coq tactics that encode common proof patterns for verification conditions generated by the system. Jack also includes a tool for generating JML specifications. This tool infers minimal preconditions for methods based on necessary null dereference and array bounds checks, and generates annotations based on developer-supplied security rules. Jack is implemented as an Eclipse IDE, with a proof obligation viewer and a Coq editor, both of which are also integrated into Eclipse.

Jack has been used to verify memory consumption of Java bytecode programs [15] and high-level security properties of Java Card programs (such as atomicity of operations within transactions) [127]. Although it is possible, in principle, to verify functional correctness using Jack, the published results so far demonstrate only the application of Jack to the verification of security properties. Unlike Jahob, Jack does not support integrated reasoning or an integrated proof language.

9.2 Interactive Theorem Provers

Interactive theorem provers, or proof assistants, are systems that allow users to define formalisms and write machine-checkable proofs about their properties. Interactive theorem provers enable the verification of algorithms and programming languages as well as logics. The general-purpose nature of these systems makes it possible to develop custom theories for the verification of executable programs. Because of

their interactive nature, it is theoretically possible to prove as strong properties using proof assistants as is possible with program verification systems like Jahob, but at the cost of a much higher level of user interaction. The environment in which the verification takes place also tends to be very different from the standard programming environment of the target language, and may require a great deal of domain expertise to use effectively.

Jahob's integrated proof language is designed to enable the proof of complex program properties within the program verification system, without leaving the context of the original program or giving up access to the full complement of automated reasoning techniques integrated into Jahob. While most interactive theorem provers operate over functional programming languages in an environment designed for proofs, Jahob's proof language is integrated into the underlying imperative programming language and naturally extends its assertion mechanism. This approach provides the developer with an accessible way of reasoning not only about the effect of executing a method but also about the intermediate states during the execution, which is often necessary when verifying complex program properties. Moreover, each proof command is able to direct the efforts of any of the automated reasoning techniques that are integrated into Jahob, requiring the user to provide only the minimum amount of guidance that the provers need to enable a successful proof.

Like many other program verification systems, Jahob also supports the use of interactive provers to discharge difficult proof obligations. The advantage of this approach is that there is, in theory, no limit on the difficulty of the properties that the system is able to prove. But in our experience, mechanically generated proof obligations often contain large numbers of assumptions that make it difficult to manipulate the resulting sequents in a proof assistant. Moreover, the presence of temporary variable names makes it difficult to map the meaning of the sequents back into the concepts of the original program. Jahob's proof language provides an alternative way of decomposing proof obligations without leaving the context of the original Java program. Its proof commands make it possible to limit the number of assumptions under consideration, and to make declarative statements about program and specification variables using existing names. The fact that these proof commands naturally translate into guarded commands suggests that they are intuitive for the verification of imperative programs.

We note that some proof assistants support primarily tactic-style proofs. Unlike declarative proofs, these proofs are scripts that must be executed to show the intermediate facts that support the proof, making them difficult to understand. Tactic-style proofs also suffer from problems of robustness and maintainability in the presence of changes to the names of intermediate variables and theorem prover tactics.

Here we survey a number of popular proof assistants, and discuss the relationship between Jahob and these systems.

9.2.1 Isabelle

Isabelle [157] is a generic proof assistant that follows the LCF (Logic for Computable Functions) approach; it supports a meta logic that enables the formalization of different calculi, including, for example, HOL [123, 122]. Isabelle supports both a tactic-

style proof language as well as the declarative language Isar [158]. Automation is provided by means of a classical reasoner that provides a collection of automatic tactics, as well as interfaces to external first-order provers.

Jahob includes both automatic and interactive interfaces to Isabelle/HOL. The former allows Isabelle to be invoked as an automatic prover and consists of a collection of Jahob-specific theories, as well a generic proof script that invokes tactics using Isabelle's command-line interface. The manual interface generates Isabelle/HOL theory files so users can interactively prove sequents that do not prove automatically. Jahob reads these theory files to extract proved lemmas, which it matches against generated proof obligations using a matching algorithm that can match proved sequents in the presence of renamed variables, unnecessary assumptions, and different assumption ordering.

There have been a number of efforts to verify properties of Java and Java programs using Isabelle. Bali [80] is a collection of theories implemented in Isabelle that model many of the features of the Java Card programming language. MicroJava [79] is a reduced model of Java Card that includes the bytecode, bytecode verifier, and compiler. Both Bali and MicroJava include proofs of various properties of the modeled systems, and can be used to prove properties of Java Card programs. Isabelle has also been used to prove certain soundness properties of a model of a Java bytecode verifier [131].

There have also been efforts to verify data structures using Isabelle, though these data structures are not implemented in Java. Isabelle has been used to implement and verify purely functional data structures such as a binary search tree with a map interface [81] and an AVL tree with a set interface [124]. In the Verisoft project, researchers have developed Isabelle/HOL proofs of functional correctness for a doubly-linked list implemented in C0, a C-like programming language [3].

Isabelle has also been used to verify the full functional correctness of a microkernel written in C and ARMv6 assembler [78]. The proof required 200,000 lines of Isabelle script and 11 person years, not including the code, tools, and Isabelle extensions on which the proof is based.

9.2.2 Coq

Coq is a proof assistant [24, 36] for the calculus of inductive constructions. It supports the formalization and proofs of both mathematical theorems and software specifications. From the latter, Coq can be used to extract certified functional programs in Objective Caml, Haskell, or Scheme. Coq supports tactic-style proofs as well as a tactic language for writing user-defined tactics. Although Coq is designed primarily for interactive use, it includes decision procedures and semi-decision procedures for some theories including proposition calculus and Presburger arithmetic. It also supports a language, called Ltac, for implementing decision procedures in Coq. Jahob contains an interface to Coq for interactively proving sequents that do not prove automatically.

While Coq can only be used to write functional programs that can be proved to terminate, Ynot [118, 37] is an extension to Coq that enables users to write possibly non-terminating, imperative programs using monads. Executable code for the

resulting programs can be produced using Coq's extraction mechanism. After the publication of our full functional verification results [163], the most recent version of Ynot was used to verify a collection of data structure implementations [37]. The verified data structures include hash tables, binary search trees, as well as a port of a Jahob association list example that we had previously verified. Proofs written using Coq tactics were supplemented by automation in the form of a simplification procedure for higher-order separation logic. Simple data structures, such as association lists, that require no proof commands in Jahob to verify, required, in Ynot, about twice as many lines of proof and tactics as implementation. More complex data structures, such as hash tables, required fewer lines of proof and tactics in Ynot than in Jahob, but this is partly due to different methods of accounting. As measured in [37], each line of code, proof, or tactic in Ynot may perform multiple operations, while for Jahob, we count each operation as a single logical line of code or proof. When considered as a function of the number of lines of implementation, Ynot required about three times as many lines of proof and tactics as code (to verify a hash table), while Jahob required only about twice as many.

Coq has also been used to implement and verify purely functional data structures including binary search trees, sorted lists, red-black trees, AVL trees, and finger trees [2].

9.2.3 ACL2

The ACL2 (A Computational Logic for Applicative Common Lisp) system [73, 1] consists of a programming language for modeling computer systems and an interactive proof tool for proving properties about these models. It is the successor to the Boyer-Moore theorem provers Nqthm and its interactive enhancement Pc-Nqthm [29]. The programming language for ACL2 is an applicative (purely functional) subset of Common Lisp, which means that ACL2 models can be executed as Common Lisp programs. The logic used is a first-order, quantifier-free logic of total recursive functions, with induction up to ϵ_0, and extension principles for ensuring consistency of the extended logic. The ACL2 theorem prover uses rewriting, decision procedures, and other standard search proof techniques when attempting to prove a formula. If the prover is unable to find a proof, the user can provide guidance by directing the system to prove key lemmas, which is similar to the use of the note command in Jahob without from clauses. ACL2 also includes an interface that may be used to integrate external proof tools [133, 74]. Currently integrated tools, which do not yet use the new interface, include the Cadence SMV model checker [109], SAT solvers Zchaff and Minisat [134, 165, 53], SixthSense [147, 112], and UCLID [105, 90].

ACL2 has been used to develop models of the JVM, which can be used to verify properties of Java byte code programs as well as properties of the JVM model [114, 101, 102]. This approach has been used to verify several examples, including recursive and iterative implementations of factorial, a functional implementation of insertion sort, as well as a proof of progress for a multi-threaded example involving mutual exclusion. The challenge in applying this approach is that proofs must necessarily involve not only the target program, but also the JVM model, increasing the difficulty

of the proof task. Once proved, lemmas about the JVM may be reused for subsequent verifications, but the up-front cost of verification appears to be much more substantial than when using Jahob or other program verification systems.

ACL2 has also been used to verify correctness properties of many other examples, including models of processor components [31], and functional data structures such as lists that implement a set interface [113, 44] and record structures [30, 75, 45].

9.2.4 KIV

KIV (Karlsruhe Interactive Verifier) [9, 135] is an interactive tool designed to support formal systems development, including the verification of Java Card programs [151]. Properties of interest are expressed in a dynamic logic, and proved using a system of tactics, which also supports automated proof search.

KIV has been used to verify the correctness of a Java Card implementation of Mondex, a protocol for electronic purses [62, 66, 63]. Although the implementation of Mondex verified in this case study includes the use of a recursive, linked data representation in the form of a Document class, the verification focuses on high-level security properties, and does not cover data abstraction or invariants. The implementation of the message encoding protocol also required that a fixed bound be placed on the recursion depth to ensure correctness, as Document objects were not guaranteed to be acyclic.

9.3 Data Structure Verification

Program verification systems and interactive theorem provers are not the only types of systems that have been applied to the data structure verification problem. Here we survey the areas of shape analysis, separation logic, type systems, and decidable logics as they relate to data structure verification. There is some overlap in these areas as technologies converge, as evidenced by systems such as SpaceInvader [161] and THOR [104], which use separation logic based shape analysis.

9.3.1 Shape Analysis

Shape analysis is a type of static analysis for tracking aliasing and structural properties of programs. It has traditionally been used to verify properties of imperative linked data structures, though these properties are generally limited to shape properties, and do not encompass full functional correctness properties such as data structure content. While shape analysis is very powerful, issues of scalability have limited its applicability. The approach we use in Jahob allows shape analysis to be combined with other reasoning techniques and to be used in a modular way. Data structure implementations can be verified using shape analysis and other techniques. The verified interfaces can then be used to reason about data structure clients, enabling verification of richer properties and avoiding the need for whole program analysis.

Jahob contains an implementation of symbolic shape analysis [128, 159, 160], called Bohne, which generalizes predicate abstraction to perform shape analysis. It has been used to verify operations on various data structures including lists, trees, and arrays [160]. The analysis successfully inferred loop invariants and proved the full functional correctness of operations that insert elements into data structures that implement a set interface. While we did not use Bohne for the examples presented in this thesis, we were able to verify more difficult properties using Jahob's interface to the MONA [67] decision procedure in combination with the other prover interfaces.

Here we describe two well-known systems, TVLA [100] and PALE [111], as well as several more recent shape analysis research systems that have been applied to data structures.

- **TVLA** [143, 144, 100, 25, 26] is a generic framework for abstract interpretation, and, in particular, shape analysis. TVLA has been used to implement shape analysis extended with ordering information for proving partial correctness properties (shape and sortedness) of C procedures that manipulate and sort linked lists [99]. It has also been used to implement shape analyses for verifying singly-linked list and binary search tree implementations with respect to a set interface [136], not including size properties. One potential source of unsoundness in these results is the use of manually specified and unverified rules for updating instrumentation predicates. TVLA has since been extended with techniques for soundly generating predicate maintenance formulas for some classes for analyses [137]. This approach has been used to verify partial correctness properties of a number of data structure operations, specifically, shape properties of singly and doubly-linked lists and binary search trees, and sortedness of singly-linked list sort operations. More recent results include the verification of these and other partial correctness properties of list and tree operations [25, 26], including memory safety and termination for some examples.

- **PALE** [111] (Pointer Assertion Logic Engine) is a framework for verifying partial specifications of programs encoded in monadic second-order logic. It requires method contracts and loop invariants, and uses the MONA [67] decision procedure to decide the resulting Hoare triples. As in TVLA, instrumentation predicates can be used to augment the analysis with ordering information. It has been used to verify partial correctness properties of singly and doubly-linked lists, red-black search trees, and threaded trees, including sortedness of a singly-linked list bubble sort, and the red-black tree invariant for operations on red-black trees. Other properties verified include the lack of null dereferences and shape properties.

- Berdine *et al.* [18] describe a shape analysis based on separation logic for composite data structures (e.g. a cyclic linked list of acyclic linked lists). The analysis uses higher-order inductive predicates for linear data structures to enable the synthesis of new predicates for composite data structures. It has been applied to a number of list manipulating routines from a Windows IEEE 1394

(firewire) device driver written in C, and was able to prove the memory safety of some, and identify several previously unknown bugs in others.

- Chang, Rival and Necula [35] describe a lightweight automatic shape analysis based on separation logic. The analysis infers the necessary shape invariants from data structure invariants provided by the developer in the form of data structure checking code such as that used in testing. A prototype system has been used to verify structural (i.e. shape) properties of list operations, binary search tree find, and a Linux device driver that uses a list of doubly-linked lists. In each case, the system verified that the operations preserved the structural invariants inferred from the data structure checking code.

- Guo, Vachharajani and August [64] describe an interprocedural shape analysis algorithm based on separation logic. It does not require loop invariants or method contracts, and is also able to infer recursive shape invariants for data structures with a tree-like backbone by analyzing the loop that constructs the data structure (though the algorithm may not always succeed). The analysis has been applied to verify shape properties of C programs that manipulate list and tree data structures.

- **SpaceInvader** [161] is a tool that uses a separation logic based shape analysis to verify the absence of pointer safety violations in device drivers that manipulate shared singly and doubly-linked lists. It has been used to prove the absence of pointer safety violations in examples, including one that contains over 10,000 lines of code.

- **THOR** [104, 103] (Tool for Heap-Oriented Reasoning) is a tool that combines a shape analysis based on separation logic with arithmetic reasoning to verify memory safety of programs. The shape analysis that THOR uses is able to reason about doubly-linked lists, while the arithmetic reasoning supported encompasses stack-based integers, integers in the heap, and the length of lists. The analysis proceeds in two phases. In the first phase, THOR uses symbolic execution to explore all paths of the program. For programs that can be shown to be memory safe using shape analysis, only the first phase is necessary. Otherwise, THOR generates a purely arithmetic program from the results of the shape analysis. If this program can be shown to not reach a designated error state, then the original program is guaranteed to be memory safe. The arithmetic program, which is output as C code by default, must then be checked using an external tool such as BLAST or ARMC.

9.3.2 Separation Logic

Separation logic is a program logic for reasoning about mutable shared data structures [138]. It is based on the idea that the program heap can be separated into disjoint regions. Special operators in the logic make it possible to make statements about a locally updated region without explicit frame conditions for disjoint portions

of the heap, enabling compact specifications and more straightforward reasoning. Separation logic can also be used to enforce abstractions by eliding properties of private state from method specifications. While separation logic shows great promise for dealing with issues of aliasing in the presence of destructive updates, current systems, with the exception of Ynot, are generally able to verify only partial correctness properties, and for languages designed for the particular analysis. In basing Jahob's specification language on classical higher-order logic, we were able to take advantage of the many off-the-shelf provers not yet available for separation logic. As the field matures, we expect that more automated tools will be available. Here we briefly discuss several systems based on separation logic that have been used to verify properties of imperative linked data structures.

- **Smallfoot** [20, 19] is an automatic tool for checking lightweight assertions in separation logic using symbolic execution. The input language contains first-order procedures with reference and value parameters, and primitives for allocating, deallocating, mutating and reading heap cells. Users write preconditions, postconditions, and loop invariants for methods in a specification language based on separation logic, from which the system generates verification conditions, then discharges them by symbolic execution. While the specification language is not sufficiently rich for full functional correctness verification, it has been used to verify shape properties of operations on data structures including disposal and copying of a tree, appending two linked lists, and adding and removing nodes from a double-ended queue implemented using an xor-linked list.

- A prototype system by Nguyen, David, Qin and Chin uses user-defined inductive predicates in separation logic to verify both shape and size properties of data structures [121, 120]. In this approach, users define predicates that capture the shape and size properties of the target data structure, write pre- and postconditions for methods and loops in terms of these predicates, and state auxiliary relations between predicate definitions in the form of lemmas. The system operates on a strongly-typed, simple imperative language. The verification conditions generated by the system are automatically discharged by the internal entailment proving procedure, using the external tools Omega calculator and CVC Lite [13] as arithmetic solvers. The system has been used to prove shape, size, and sortedness properties on data structures, but not full functional correctness.

- **Bigtoe** [108] is a certified verifier for a decidable fragment of separation logic with Presburger arithmetic, which is able to automatically prove separation logic triples. It is implemented and verified in Coq, and can be executed either as a tactic, or a stand-alone OCaml program generated via Coq's extraction mechanism. The system operates on a language with assignment, lookup, mutation, allocation, deallocation, while loops, and if-then-else. It uses both forward and backward reasoning to automatically discharge generated verification

conditions. The extracted OCaml implementation has been used to prove shape properties of simple list operations.

Separation logic has also been used in tools such as MUTANT [21] to prove termination of loops that manipulate linked lists.

9.3.3 Type Systems

While type systems have traditionally been used to enforce lightweight correctness properties of programs, more sophisticated type systems such as dependent types can be used to enforce stronger correctness properties such as sortedness and size properties of data structures. Systems that support dependent types in combination with proofs, such as Coq, PVS, and ATS [41], enable correctness properties to be encoded as types, with the ability to write the proofs necessary for type checking. These approaches offer types as an alternative specification mechanism.

Standalone type-based approaches have also been used to verify rich correctness properties of data structure implementations. Stardust [52] is a type checker for a type system with datasort and index refinement for functional programs, in which users provide type annotations, along with subsort relations, that are checked automatically by the system. It has been used to verify data structure invariants including red black tree invariants of insertions and deletions into a functional red black tree, though these invariants do not encompass correctness of the tree contents.

Recursive and polymorphic type refinements have also been used to verify properties of data structure implementations [76]. In this approach, users provide a program written in NanoML (a functional ML-like language), type specifications, measure definitions, and logical qualifiers (where measures are recursive functions that compute desired correctness properties, and logical qualifiers the predicates of interest), which are automatically checked by the type checker. Verified examples include functional map and heap implementations with respect to a set interface, and the correctness of list sort. While this and other type-based approaches offer, in general, a higher level of automation than program verification approaches such as Jahob, the properties that have been verified have not extended to more sophisticated interfaces such as maps, and do not address the difficulties associated with aliasing and destructive updates that are essential to verifying imperative programs.

9.3.4 Decidable Logics

Many decidable logics exist which can be used to express important properties of data structures [67, 17, 71, 146, 162, 50, 84]. In isolation, these logics enable automated verification, but only of a limited class of properties. Jahob's integrated reasoning approach enables the combination of decision procedures for such logics with other reasoning techniques, thereby extending their applicability to a richer class of properties. Jahob currently incorporates decision procedures for BAPA [84] (Boolean Algebra with Presburger Arithmetic) for deciding formulas involving cardinality, and

the external decision procedure MONA [67] for monadic second order logic over trees, for formulas involving shape properties.

9.4 Finitization and Automated Testing

Jahob and other program verification systems based on theorem proving are able to verify the correctness of a program for all possible executions. In contrast, verification techniques based on finitization (such as testing [107, 28, 155, 150], model checking [72, 8, 7, 154, 140, 141, 46], and bounded verification [47, 48, 49]) offer a higher level of automation than program verification techniques, but are able to verify correctness for only finitely many executions. Nevertheless, these techniques are useful for detecting program errors, and can be used in conjunction with program verification techniques to ensure program correctness. In particular, these techniques can be efficient in early stages of development, for finding errors in the program and specification, or for software components where the level of assurance required is not sufficient to warrant the higher cost of verification. Systems that support techniques based on finitization in addition to theorem proving include the proof assistants Isabelle [65], PVS [125], and ACL2 [133, 74].

9.5 Summary

Data structure verification is a difficult problem that has been explored in many areas of research. Program verification systems, interactive theorem provers, shape analysis, type systems, and logics have all been applied to this problem, but due to the difficult issues caused by aliasing in the presence of destructive updates, the large majority of these systems have focused only on partial correctness properties, or on functional implementations for which verification is more tractable. To the best of our knowledge, Jahob is the first system to have been used to successfully verify the full functional correctness of a substantial collection of imperative linked data structures. Jahob's integration of a proof language as well as a diversity of automated reasoning techniques enables a unique combination of automation and control that allows us to successfully verify the complex correctness properties that arise in imperative data structure verification. Of subsequent results, only Ynot, an extension to the Coq proof assistant, has been used to verify properties and programs of comparable difficulty. Although the results that we have obtained are, in theory, possible using other systems—specifically, proof assistants such as Coq—our integration of a declarative proof language within the program verification system offers a unique way of embedding proofs in the familiar context of the original programming language not possible in interactive theorem provers that operate in an environment designed for proofs. This approach not only enables proofs to be naturally embedded within imperative code, but also gives the user full access to all the automated reasoning techniques that are integrated into the Jahob program verification system.

Chapter 10

Conclusion

In this thesis, we have described the Jahob program verification system and our experience using it to verify the full functional correctness of imperative linked data structures. Through the use of integrated reasoning, we were able to leverage the strengths of a diversity of automated reasoning techniques to prove the full functional correctness of a collection of imperative linked data structure implementations. We also made use of Jahob's integrated proof language, which enabled us to provide the guidance needed by the provers to establish the correctness of properties that are otherwise beyond the capabilities of the automated techniques. The interfaces that we have verified capture all the properties relevant to the data structure client (with the exception of properties involving execution time and/or memory consumption) while preserving abstraction. Our experience shows that integrated reasoning, together with Jahob's proof language, was effective in proving full functional correctness properties that have traditionally been beyond the scope of program verification systems without the use of external proof assistants.

Our integrated reasoning approach is also supported by our empirical results. All but one of the data structures verified required the use of more than one automated reasoning technique, demonstrating that a diversity of automated reasoning techniques can be more effective than any single prover. While examples that verified without any proof commands—such as our association list implementation of a map interface—highlight the effectiveness of integrated reasoning in discharging verification conditions for non-trivial examples, the verification of even more sophisticated data structures would not have been possible without the use of our integrated proof language. The proof language enabled us to provide the guidance that the provers needed to successfully verify difficult verification conditions that the automated reasoning techniques were unable to discharge without guidance.

Our results also suggest that, although programs that directly manipulate linked data structures are difficult to verify, programs that do so indirectly—by invoking methods that perform the actual manipulation—may be verified with little or no user interaction. The majority of the proof commands used in the verification occur in methods that directly manipulate the data structure state. With few exceptions, we were able to verify methods that indirectly modify the data structure state with either few or no proof commands. This somewhat surprising result further supports the

verification of data structure libraries, where the verification cost may be amortized over many uses, as it suggests that library clients may be easier to verify, and that our approach should scale well to larger programs that indirectly manipulate data structures.

10.1 Future Directions

Our experience using Jahob to verify imperative linked data structures has also given us insight into promising directions for future work. One such direction is in improving the ability of the system to handle instantiable data structures. Modifying an existing data structure implementation in Java to enable multiple instances can be as easy as removing a few static keywords. But verifying an instantiable data structure can require significantly more effort than its static counterpart due to issues of aliasing. Most instantiable data structures do not share state between instances, and the verification effort for this common case should ideally be no more than the effort to verify a single static instance. But in practice, the formulas that need to be proved to verify the lack of sharing between instances greatly exacerbates the difficulty of the proof task. While the large majority of the data structures we verified are instantiable, the two data structures that use the MONA decision procedure (the binary search tree and circular list) are not. This is because the way in which Jahob currently translates formulas for MONA makes it much easier to verify static data structures than instantiable ones. We would like to explore ways to improve this translation. We would also like to incorporate other techniques for handling shape properties—including techniques based on separation logic and abstract interpretation—that may be more naturally suited for addressing issues of aliasing. We have already done some work in applying abstract interpretation to this problem, with promising preliminary results.[1]

Another interesting direction for future work is in applying Jahob to verify properties beyond functional properties. Examples include system-specific properties such as time and memory consumption, termination, and safety and security properties. We have a prototype system for verifying safety properties using Jahob by automatically translating properties that can be encoded as temporal logic formulas into Jahob specifications, but so far we have applied our system only to small examples.

Other interesting directions include applying Jahob to larger programs, and to parallel programs. Our results suggest that our approach should scale, especially to larger programs that indirectly manipulate data structures, and we expect verifying these programs to be straightforward in most cases. But larger programs may also give rise to more complex inter-class dependencies than we have seen in our examples. It is an open question as to whether the techniques we have used to verify data structures can also handle these dependencies. We are also interested in extending Jahob to verify properties of parallel programs. Approaches based on assume-guarantee reasoning enable properties of parallel programs to be expressed and verified in an elegant and

[1] Jahob can automatically determine whether a static data structure can be made instantiable by checking whether the implementation correctly encapsulates its private state.

modular way [98], making them natural candidates for further development within Jahob's existing assume-guarantee infrastructure.

Last but not least, we are interested in exploring how verified data structures can be used as the basis for a new class of static program analyses. Because verified specifications soundly capture the behavior of called methods, it may be possible to use these specifications as method summaries, thereby avoiding interprocedural analysis and improving scalability. It may also be possible to improve the precision of some program analyses using this technique—by purposefully omitting from the specifications information about the data structure that is not intended to be exposed to the client. Differences in the concrete structure of data behind the abstraction barrier may obscure an analysis' ability to recognize structures that are semantically equivalent, but verified interfaces can present only the important semantic information, improving the ability of the analysis to extract the desired results. We have done some preliminary work in applying this technique to commutativity analysis [139], and are interested in continuing to pursue research in this area.

10.2 Summary

Data structure verification is an area of research that has inspired numerous automated and interactive reasoning techniques. In Jahob, we take a hybrid approach that enables both the use of state of the art automated reasoning techniques as well as the suitable application of developer insight in the verification process. Where traditional program verification approaches have focused only on partial correctness properties, the application of integrated reasoning and the use of the declarative proof language in our system has enabled us to verify the full functional correctness of a collection of imperative linked data structures implementations. The programs and properties we have verified exceed what previous techniques have been able to achieve, and have since been matched only by approaches based on interactive theorem proving. The results we have obtained using our system support our hybrid approach. They also suggest a variety of new directions in which we could extend our work—to programs and properties beyond data structure verification, and to new program analyses based on verified interfaces. Such program analyses have the potential to improve on existing analyses in both scalability and precision. By demonstrating the feasibility of full functional verification for imperative linked data structures, we hope that our results will inspire further research in developing the tools and techniques that enable the verification of sophisticated properties in larger programs.

Appendix A

Soundness Proofs

Using the proof methodology described in Section 4.2 of Chapter 4, we prove the soundness of the translation for each of the proof language commands given in Figure 3-5, and for assert. Figures A-1, A-2, and A-3 present these soundness proofs. The simplicity of these proofs illustrates the methodological advantages of defining proof commands through a translation into guarded command language.

p	$\mathsf{wlp}(\llbracket p \rrbracket, H)$
$p_1 ; p_2$	$\mathsf{wlp}(\llbracket p_1 ; p_2 \rrbracket, H)$ $= \mathsf{wlp}((\llbracket p_1 \rrbracket ; \llbracket p_2 \rrbracket), H)$ $= \mathsf{wlp}(\llbracket p_1 \rrbracket, \mathsf{wlp}(\llbracket p_2 \rrbracket, H))$ $\rightarrow \mathsf{wlp}(\llbracket p_2 \rrbracket, H)$ $\rightarrow H$
assert F	$\mathsf{wlp}(\llbracket \mathsf{assert}\ F \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F), H)$ $= F \wedge H$ $\rightarrow H$
note F	$\mathsf{wlp}(\llbracket \mathsf{note}\ F \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F ; \mathsf{assume}\ F), H)$ $= F \wedge (F \rightarrow H)$ $\rightarrow H$
localize in $(p ; \mathsf{note}\ F)$	$\mathsf{wlp}(\llbracket \mathsf{localize\ in}\ (p ; \mathsf{note}\ F) \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{skip} \,[]\, (\llbracket p \rrbracket ; \mathsf{assert}\ F ; \mathsf{assume\ false})) ;$ $\quad \mathsf{assume}\ F), H)$ $= (F \rightarrow H) \wedge \mathsf{wlp}(\llbracket p \rrbracket, F)$ $\rightarrow (F \rightarrow H) \wedge F$ $\rightarrow H$

Figure A-1: Soundness Proofs for Translation of Proof Language Commands (continued in Figure A-2)

p	$\mathsf{wlp}(\llbracket p \rrbracket, H)$
$\mathsf{mp}\ (F \to G)$	$\mathsf{wlp}(\llbracket \mathsf{mp}\ (F \to G) \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F\,;\mathsf{assert}\ (F \to G)\,;\mathsf{assume}\ G), H)$ $= F \wedge (F \to G) \wedge (G \to H)$ $\to G \wedge (G \to H)$ $\to H$
$\mathsf{assuming}\ F\ \mathsf{in}\ (p\,;\mathsf{note}\ G)$	$\mathsf{wlp}(\llbracket \mathsf{assuming}\ F\ \mathsf{in}\ (p\,;\mathsf{note}\ G) \rrbracket, H)$ $= \mathsf{wlp}(((\mathsf{skip}\ [](\mathsf{assume}\ F\,;$ $\llbracket p \rrbracket\,;\mathsf{assert}\ G\,;\mathsf{assume\ false}))\,;$ $\mathsf{assume}\ (F \to G)), H)$ $= ((F \to G) \to H) \wedge (F \to \mathsf{wlp}(\llbracket p \rrbracket, G))$ $\to ((F \to G) \to H) \wedge (F \to G)$ $\to H$
$\mathsf{cases}\ \vec{F}\ \mathsf{for}\ G$	$\mathsf{wlp}(\llbracket \mathsf{cases}\ \vec{F}\ \mathsf{for}\ G \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F_1 \vee \ldots \vee F_n\,;$ $\mathsf{assert}\ (F_1 \to G)\,;\ldots;\mathsf{assert}\ (F_n \to G)\,;$ $\mathsf{assume}\ G), H)$ $= (F_1 \vee \ldots \vee F_n) \wedge (F_1 \to G) \wedge \ldots \wedge$ $(F_n \to G) \wedge (G \to H)$ $\to G \wedge (G \to H)$ $\to H$
$\mathsf{showCase}\ i\ \mathsf{of}\ F_1 \vee \ldots \vee F_n$	$\mathsf{wlp}(\llbracket \mathsf{showCase}\ i\ \mathsf{of}\ F_1 \vee \ldots \vee F_n \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F_i\,;\mathsf{assume}\ F_1 \vee \ldots \vee F_n), H)$ $= F_i \wedge ((F_1 \vee \ldots \vee F_n) \to H)$ $\to H$
$\mathsf{byContradiction}\ F\ \mathsf{in}\ p$	$\mathsf{wlp}(\llbracket \mathsf{byContradiction}\ F\ \mathsf{in}\ p \rrbracket, H)$ $= \mathsf{wlp}(((\mathsf{skip}\ [](\mathsf{assume}\ \neg F\,;$ $\llbracket p \rrbracket\,;\mathsf{assert\ false}\,;\mathsf{assume\ false}))\,;$ $\mathsf{assume}\ F), H)$ $= (F \to H) \wedge ((\neg F) \to \mathsf{wlp}(\llbracket p \rrbracket, \mathsf{false}))$ $\to (F \to H) \wedge ((\neg F) \to \mathsf{false})$ $\to H$
$\mathsf{contradiction}\ F$	$\mathsf{wlp}(\llbracket \mathsf{contradiction}\ F \rrbracket, H)$ $= \mathsf{wlp}((\mathsf{assert}\ F\,;\mathsf{assert}\ \neg F\,;\mathsf{assume\ false}), H)$ $= F \wedge \neg F$ $\to H$

Figure A-2: Soundness Proofs for Translation of Proof Language Commands (continued from Figure A-1)

p	$\mathsf{wlp}(\llbracket p \rrbracket, H)$
witness $\vec{t}$ for $\exists \vec{x}.F$	$\mathsf{wlp}(\llbracket \text{witness } \vec{t} \text{ for } \exists \vec{x}.F \rrbracket, H)$ $= \mathsf{wlp}((\text{assert } F[\vec{x} := \vec{t}]\,;\text{assume } \exists \vec{x}.F), H)$ $= F[\vec{x} := \vec{t}] \wedge ((\exists \vec{x}.F) \rightarrow H)$ $\rightarrow (\exists \vec{x}.F) \wedge ((\exists \vec{x}.F) \rightarrow H)$ $\rightarrow H$
pickWitness $\vec{x}$ for F in $(p\,;\text{note}\,G)$ (where $\vec{x}$ is not free in G)	$\mathsf{wlp}(\llbracket \text{pickWitness } \vec{x} \text{ for } F \text{ in } (p\,;\text{note}\,G) \rrbracket, H)$ $= \mathsf{wlp}(((\text{skip } [] (\text{assert } \exists \vec{x}.F\,;\text{havoc } \vec{x}\,;$ $\text{assume } F\,;$ $\llbracket p \rrbracket\,;\text{assert } G\,;\text{assume false}))\,;$ $\text{assume } G), H)$ $= (G \rightarrow H) \wedge \exists \vec{x}.F \wedge$ $\forall \vec{x}.(F \rightarrow \mathsf{wlp}(\llbracket p \rrbracket, G))$ $\rightarrow (G \rightarrow H) \wedge \exists \vec{x}.F \wedge \forall \vec{x}.(F \rightarrow G)$ $\rightarrow (G \rightarrow H) \wedge G$ $\rightarrow H$
pickAny $\vec{x}$ in $(p\,;\text{note}\,G)$	$\mathsf{wlp}(\llbracket \text{pickAny } \vec{x} \text{ in } (p\,;\text{note}\,G) \rrbracket, H)$ $= \mathsf{wlp}(((\text{skip } [] (\text{havoc } \vec{x}\,;$ $\llbracket p \rrbracket\,;\text{assert } G\,;\text{assume false}))\,;$ $\text{assume } \forall \vec{x}.G), H)$ $= ((\forall \vec{x}.G) \rightarrow H) \wedge \forall \vec{x}.\mathsf{wlp}(\llbracket p \rrbracket, G)$ $\rightarrow ((\forall \vec{x}.G) \rightarrow H) \wedge \forall \vec{x}.G$ $\rightarrow H$
induct F over n in p	$\mathsf{wlp}(\llbracket \text{induct } F \text{ over } n \text{ in } p \rrbracket, H)$ $= \mathsf{wlp}(((\text{skip } [] (\text{havoc } n\,;\text{assume } 0 \leq n\,;$ $\llbracket p \rrbracket\,;\text{assert } F[n := 0]\,;$ $\text{assert } (F \rightarrow F[n := n{+}1])\,;$ $\text{assume false}))\,;$ $\text{assume } \forall n.(0 \leq n \rightarrow F)), H)$ $= ((\forall n.(0 \leq n \rightarrow F)) \rightarrow H) \wedge$ $\forall n.(0 \leq n \rightarrow \mathsf{wlp}(\llbracket p \rrbracket, (F[n := 0] \wedge$ $(F \rightarrow F[n := n{+}1]))))$ $\rightarrow ((\forall n.(0 \leq n \rightarrow F)) \rightarrow H) \wedge$ $\forall n.(0 \leq n \rightarrow (F[n := 0] \wedge$ $(F \rightarrow F[n := n{+}1]))))$ $\rightarrow ((\forall n.(0 \leq n \rightarrow F)) \rightarrow H) \wedge$ $\forall n.(0 \leq n \rightarrow F)$ $\rightarrow H$

Figure A-3: Soundness Proofs for Translation of Proof Language Commands (continued from Figure A-2)

Bibliography

[1] ACL2. http://www.cs.utexas.edu/users/moore/acl2/, last accessed October 2009. 191

[2] Coq web site. http://www.lix.polytechnique.fr/coq/contribs/bycat.html, last accessed October 2009. User Contributions. 191

[3] Verisoft project. `http://www.verisoft.de`, last accessed September 2009. 190

[4] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Applied Logic Series, Vol. 27. Springer, 2nd edition, 2002. 75

[5] David Aspinall. Proof General: A generic tool for proof development. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '00*, volume 1785 of *Lecture Notes in Computer Science*, pages 38–43, 2000. 83

[6] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Springer, 1998. 79

[7] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI '01: Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001. 16, 197

[8] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL '02: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, 2002. 197

[9] Michael Balser, Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Andreas Thums. Formal system development with KIV. In *Fundamental Approaches to Software Engineering: Proceedings of the Third International Conference, FASE 2000*, volume 1783 of *Lecture Notes in Computer Science*, pages 363–366, 2000. 192

[10] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs.

In *Proceedings of the Third International Symposium on Formal Methods for Components and Objects, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2006. 186

[11] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004. Special Issue: ECOOP 2003 Workshop on Formal Techniques for Java-like Programs. 41, 42, 66, 183, 186

[12] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, 2005. 41, 183, 186

[13] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518, 2004. 195

[14] Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. JACK — a tool for validation of security and behaviour of Java applications. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects, FMCO 2006*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174, 2007. 15, 69, 183, 188

[15] Gilles Barthe, Mariela Pavlova, and Gerardo Schneider. Precise analysis of memory consumption using program logics. *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, SEFM 2005*, pages 86–95, 2005. 188

[16] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Artificial Intelligence*. Springer Berlin/Heidelberg, 2007. 41, 183, 186

[17] Michael Benedikt, Thomas Reps, and Mooly Sagiv. A decidable logic for describing linked data structures. In *Proceedings of the 8th European Symposium on Programming, ESOP 1999*, volume 1576 of *Lecture Notes in Computer Science*, pages 2–19, 1999. 196

[18] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192, 2007. 193

[19] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of the Third Asian Symposium on Programming Languages and Systems, APLAS 2005*, volume 3780 of *Lecture Notes in Computer Science*, pages 52–68, 2005. 195

[20] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the Third International Symposium on Formal Methods for Components and Objects, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137, 2006. 195

[21] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proceedings of the 18th International Conference on Computer Aided Verification, CAV 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 386–400, 2006. 196

[22] Joachim van den Berg and Bart Jacobs. The LOOP compiler for java and JML. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '01*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312, 2001. 69, 183, 187

[23] Joachim van den Berg, Bart Jacobs, and Erik Poll. Formal specification and verification of JavaCard's Application Identifier Class. In *Proceedings of the First International Workshop on Java on Smart Cards: Programming and Security, JavaCard 2000*, volume 2041 of *Lecture Notes in Computer Science*, pages 137–150, 2001. 187

[24] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development—Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. 16, 30, 60, 190

[25] Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In *Proceedings of the 19th International Conference on Computer Aided Verification, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 221–225, 2007. 193

[26] Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. Technical Report TR-2007-01-01, Tel Aviv University, 2007. 193

[27] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin Rinard. Using first-order theorem provers in the Jahob data structure verification system. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2007*, volume 4349 of *Lecture Notes in Computer Science*, pages 74–88, 2007. 48, 59

[28] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA '02: Proceedings of the*

2002 International Symposium on Software Testing and Analysis, pages 123–133, 2002. 197

[29] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. The Boyer-Moore theorem prover and its interactive environment. *Computers and Mathematics with Applications*, 29(2):27–62, January 1995. 191

[30] Bishop Brock. *defstructure for ACL2 Version 2.0.* Computational Logic, Inc., December 1997. 192

[31] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design, FMCAD '96*, volume 1166 of *Lecture Notes in Computer Science*, pages 275–293, 1996. 192

[32] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In *Proceedings of the FME 2003: International Symposium of Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439, 2003. 183, 188

[33] Lilian Burdy, Marieke Huisman, and Mariela Pavlova. Preliminary design of BML: A behavioral interface specification language for Java bytecode. In *Fundamental Approaches to Software Engineering: Proceedings of the 10th International Conference, FASE 2007*, volume 4422 of *Lecture Notes in Computer Science*, pages 215–229, 2007. 188

[34] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects, FMCO 2006*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363, 2006. 183, 185

[35] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *Proceedings of the 14th Annual International Static Analysis Symposium, SAS 2007*, volume 4634 of *Lecture Notes in Computer Science*, pages 384–401, 2007. 194

[36] Adam Chlipala. Certified programming with dependent types. http://adam.chlipala.net/cpdt/html/toc.html, January 2009. 190

[37] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 79–90, 2009. 190, 191

[38] Stephen Chong and Radu Rugina. Static analysis of accessed regions in recursive data structures. In *Proceedings of the 10th Annual International Static Analysis Symposium, SAS 2003*, volume 2694 of *LNCS*, pages 463–482, 2003. 16

[39] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an internet voting tally system. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128, 2005. 41, 183, 185, 186

[40] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter 6, pages 127–144. The MIT Press, 2001. 83

[41] Sa Cui, Kevin Donnelly, and Hongwei Xi. ATS: A language that combines programming with theorem proving. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems, FroCoS 2005*, volume 3717 of *Lecture Notes in Computer Science*, pages 310–320, 2005. 196

[42] Ádám Darvas and Peter Müller. Formal encoding of JML Level 0 specifications in JIVE. Technical Report 559, ETH Zurich, 2007. Annual Report of the Chair of Software Engineering. 188

[43] Ádám Darvas and Peter Müller. Faithful mapping of model classes to mathematical structures. *IET Software*, 2(6):477–499, December 2008. 188

[44] Jared Davis. Finite set theory based on fully ordered lists. In *ACL2 '04: Proceedings of the 5th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2004. 192

[45] Jared Davis. Memories: Array-like records for ACL2. In *ACL2 '06: Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and Its Applications*, pages 57–60, 2006. 192

[46] Xianghua Deng, Jooyong Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, ASE 2006*, pages 157–166, 2006. 197

[47] Greg Dennis. *A Relational Framework for Bounded Program Verification.* PhD thesis, Massachusetts Institute of Technology, July 2009. 197

[48] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 109–120, 2006. 16, 197

[49] Greg Dennis, Kuat Yessenov, and Daniel Jackson. Bounded verification of voting software. In *Proceedings of the Second International Conference on Verified Software: Theories, Tools, Experiments, VSTTE 2008*, pages 130–145, 2008. 197

[50] Jyotirmoy V. Deshmukh, E. Allen Emerson, and Prateek Gupta. Automatic verification of parameterized data structures. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '06*, volume 3920 of *Lecture Notes in Computer Science*, pages 27–41, 2006. 196

[51] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998. 15, 41, 183, 185

[52] Joshua Dunfield. *A Unified System of Type Refinements.* PhD thesis, Carnegie Mellon University, August 2007. 196

[53] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing, SAT 2003 (Selected Revised Papers)*, volume 2919 of *Lecture Notes in Computer Science*, pages 503–518, 2004. 191

[54] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003. 187

[55] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of the 19th International Conference on Computer Aided Verification, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, 2007. 41, 69, 183, 187

[56] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 2002. 15, 41, 44, 183, 185

[57] Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *POPL '01: Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 193–205, 2001. 54

[58] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32, 1967. 41

[59] Jean Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* `http://www.cis.upenn.edu/~jean/gbooks/logic.html`, revised online edition, 2003. 74

[60] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE-21: Proceedings of the 21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 167–182, 2007. 18, 59, 169

[61] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley, 2005. 41

[62] Holger Grandy, Robert Bertossi, Kurt Stenzel, and Wolfgang Reif. ASN1-light: A verified message encoding for security protocols. *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2007*, pages 195–204, 2007. 192

[63] Holger Grandy, Markus Bischof, Kurt Stenzel, Gerhard Schellhorn, and Wolfgang Reif. Verification of Mondex electronic purses with KIV: From a security protocol to verified code. In *FM 2008: Proceedings of the 15th International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 165–180, 2008. 192

[64] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 256–265, 2007. 16, 194

[65] Tobias Hamberger. Integrating theorem proving and model checking in Isabelle/IOA. Technical report, Technische Universität München, August 1999. 197

[66] Dominik Haneberg, Gerhard Schellhorn, Holger Grandy, and Wolfgang Reif. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Formal Aspects of Computing*, 20(1):41–59, 2008. 192

[67] Jesper G. Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Proceedings of the First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '95*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110, 1995. 18, 59, 169, 193, 196, 197

[68] Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of the 18th European Conference on Object-Oriented Programming, ECOOP 2004*, volume 3086 of *Lecture Notes in Computer Science*, pages 96–122, 2004. 42, 66

[69] Marieke Huisman, Bart Jacobs, and Joachim van den Berg. A case study in class library verification: Java's Vector class. *International Journal on Software Tools for Technology Transfer*, 3(3):332–352, August 2001. 187

[70] Neil Immerman, Alex Rabinovich, Tom Reps, Mooly Sagiv, and Greta Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Proceedings of the 18th International Workshop on Computer Science Logic, CSL 2004*, volume 3210 of *Lecture Notes in Computer Science*, pages 160–174, 2004. 16

[71] Neil Immerman, Alexander Rabinovich, Thomas W. Reps, Mooly Sagiv, and Greta Yorsh. Verification via structure simulation. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 236–239, 2004. 196

[72] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering Methodology*, 11(2):256–290, 2002. 197

[73] Matt Kaufmann and J S. Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997. 191

[74] Matt Kaufmann, J Strother Moore, Sandip Ray, and Erik Reeber. Integrating external deduction tools with ACL2. *Journal of Applied Logic*, 7(1):3–25, March 2009. 191, 197

[75] Matt Kaufmann and Rob Sumners. Efficient rewriting of operations on finite structures in ACL2. In *ACL2 '02: Proceedings of the Third International Workshop on the ACL2 Theorem Prover and Its Applications*, April 2002. 192

[76] Ming Kawaguchi, Patrick Rondon, and Ranjit Jhala. Type-based data structure verification. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 304–315, 2009. 196

[77] Joseph R. Kiniry, Patrice Chalin, and Clément Hurlin. Integrating static checking and interactive verification: Supporting multiple theories and provers in verification. In *Proceedings of the First IFIP TC 2/WG 2.3 Conference on Verified Software: Theories, Tools, Experiments, VSTTE 2005*, volume 4171 of *Lecture Notes in Computer Science*, pages 153–160, 2008. Revised Selected Papers and Discussions. 41, 42, 66, 185

[78] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP 2009*, 2009. 190

[79] Gerwin Klein, Tobias Nipkow, David von Oheimb, Leonor Prensa Nieto, Norbert Schirmer, and Martin Strecker. Java source and bytecode formalizations in Isabelle: μJava. http://isabelle.in.tum.de/verificard/MicroJava/document.pdf, January 2002. 190

[80] Gerwin Klein, Tobias Nipkow, David von Oheimb, Leonor Prensa Nieto, Norbert Schirmer, and Martin Strecker. Java source and bytecode formalizations in Isabelle: Bali. http://isabelle.in.tum.de/verificard/Bali/document.pdf, April 2003. 190

[81] Viktor Kuncak. Binary search trees. The Archive of Formal Proofs, `http://afp.sourceforge.net/`, April 2004. 190

[82] Viktor Kuncak. *Modular Data Structure Verification.* PhD thesis, Massachusetts Institute of Technology, February 2007. 184

[83] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12):988–1005, 2006. 16, 183, 184

[84] Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. An algorithm for deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *CADE-20: Proceedings of the 20th International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 260–277, 2005. 60, 196

[85] Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. Deciding Boolean Algebra with Presburger Arithmetic. *Journal of Automated Reasoning*, 36(3):213–239, April 2006. 18

[86] Viktor Kuncak and Martin Rinard. Existential heap abstraction entailment is undecidable. In *Proceedings of the 10th Annual International Static Analysis Symposium, SAS 2003*, volume 2694 of *Lecture Notes in Computer Science*, 2003. 16

[87] Viktor Kuncak and Martin Rinard. Towards efficient satisfiability checking for Boolean Algebra with Presburger Arithmetic. In *CADE-21: Proceedings of the 21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 215–230, 2007. 18, 60

[88] Marcel Labeth, Jörg Meyer, Peter Müller, and Arnd Poetzsch-Heffter. Formal verification of a doubly linked list implementation: A case study using the JIVE system. Technical Report 270, Fernuniversität Hagen, 2000. 188

[89] Shuvendu Lahiri and Shaz Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–182, 2008. 16

[90] Shuvendu Lahiri, Sanjit A. Seshia, and Randal E. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design, FMCAD '02*, volume 2517 of *Lecture Notes in Computer Science*, pages 142–159, 2002. 191

[91] Patrick Lam. *The Hob System for Verifying Software Design Properties.* PhD thesis, Massachusetts Institute of Technology, February 2007. 16, 183, 184

[92] Patrick Lam, Viktor Kuncak, and Martin Rinard. Crosscutting techniques in program specification and analysis. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 169–180, 2005. 183, 184

[93] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, revision 1.235 edition, July 2008. Draft. 185

[94] Gary T. Leavens, Clyde Ruby, K. Rustan M. Leino, Erik Poll, and Bart Jacobs. JML (poster session): Notations and tools supporting detailed design in Java. In *OOPSLA '00: Addendum to the 2000 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 105–106, 2000. 185

[95] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *Proceedings of the 14th European Symposium on Programming, ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, 2005. 16

[96] K. Rustan M. Leino. *This is Boogie 2*, 2008 June. Manuscript KRML 178, working draft 24 June 2008. 186

[97] K. Rustan M. Leino and Peter Müller. Using the Spec# language, methodology, and tools to write bug-free programs. http://research.microsoft.com/en-us/um/people/leino/papers/krml189.pdf, September 2009. Manuscript KRML 189, 17 September 2009. 183, 186

[98] K. Rustan M. Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222, 2009. 201

[99] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *ISSTA '00: Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 26–38, 2000. 193

[100] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Proceedings of the 7th Annual International Static Analysis Symposium, SAS 2000*, volume 1824 of *Lecture Notes in Computer Science*, pages 105–110, 2000. 193

[101] Hanbing Liu and J Strother Moore. Java program verification via a JVM deep embedding in ACL2. In *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 184–200, 2004. 191

[102] Hanbing Liu and J Strother Moore. Executable JVM model for analytical reasoning: A study. *Science of Computer Programming*, 57(3):253–274, 2005. Special Issue on Advances in Interpreters, Virtual Machines and Emulators (IVME '03). 191

[103] Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *Proceedings of the 14th Annual International Static Analysis Symposium, SAS 2007*, volume 4634 of *Lecture Notes in Computer Science*, pages 419–436, 2007. 194

[104] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. THOR: A tool for reasoning about shape and arithmetic. In *Proceedings of the 20th International Conference on Computer Aided Verification, CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 428–432, 2008. 192, 194

[105] Panagiotis Manolios and Sudarshan K. Srinivasan. Automatic verification of safety and liveness for XScale-like processor models using WEB refinements. In *Proceedings of Design, Automation and Test in Europe, DATE 2004*, pages 168–173, 2004. 191

[106] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`. 41, 183, 187

[107] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proceedings of the 16th IEEE/ACM International Conference on Automated Software Engineering, ASE 2001*, pages 22–31, 2001. 197

[108] Nicolas Marti and Reynald Affeldt. A certified verifier for a fragment of separation logic. *Computer Software*, 25(3):135–147, 2008. 195

[109] Kenneth L. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, May 1992. 191

[110] Jörg Meyer, Peter Müller, and Arnd Poetzsch-Heffter. The JIVE system—implementation description, 2000. 15, 69, 183, 188

[111] Anders Møller and Michael I. Schwartzbach. The pointer assertion logic engine. *SIGPLAN Notices*, 36(5):221–231, 2001. 193

[112] Hari Mony, Jason Baumgartner, Viresh Paruthi, Robert Kanzelman, and Andreas Kuehlmann. Scalable automated verification via expert-system guided transformations. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design, FMCAD '04*, volume 3312 of *Lecture Notes in Computer Science*, pages 159–173, 2004. 191

[113] J Strother Moore. Finite set theory in ACL2. In *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 313–328, 2001. 192

[114] J Strother Moore. Proving theorems about Java and the JVM with ACL2. In *Proceedings of the NATO Advanced Study Institute on Models, Algebras and Logic of Engineering Software*, volume 191 of *Series III: Computer and Systems Sciences*, pages 227–290. IOS Press, 2003. 191

[115] Wojciech Mostowski. Fully verified Java Card API reference implementation. In *VERIFY'07: Proceedings of the 4th International Verification Workshop*, volume 259 of *CEUR WS*, July 2007. 187

[116] Leonardo Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In *CADE-21: Proceedings of the 21st International Conference on Automated Deduction*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198, 2007. 18, 30, 59, 169

[117] Leonardo Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, 2008. 30, 59, 169

[118] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP '08: Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, 2008. 190

[119] Greg Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, June 1981. 59

[120] Huu Hai Nguyen and Wei-Ngan Chin. Enhancing program verification with lemmas. In *Proceedings of the 20th International Conference on Computer Aided Verification, CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 355–369, 2008. 195

[121] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2007*, volume 4349 of *Lecture Notes in Computer Science*, pages 251–266, 2007. 195

[122] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Berlin/Heidelberg, 2002. 16, 30, 38, 42, 60, 169, 189

[123] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer-Verlag, 2009. 189

[124] Tobias Nipkow and Cornelia Pusch. AVL trees. The Archive of Formal Proofs, `http://afp.sourceforge.net/`, March 2004. 190

[125] Sam Owre, John Rushby, and Natarajan Shankar. Integration in PVS: Tables, types, and model checking. In *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '97*, volume 1217 of *Lecture Notes in Computer Science*, pages 366–383, 1997. 197

[126] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987. 74

[127] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman, and Jean-Louis Lanet. Enforcing high-level security properties for applets. In *Proceedings of the 6th International Conference on Smart Card Research and Advanced Applications (CARDIS)*, volume 153 of *IFIP International Federation for Information Processing*, pages 1–16, 2004. 188

[128] Andreas Podelski and Thomas Wies. Boolean heaps. In *Proceedings of the 12th Annual International Static Analysis Symposium, SAS 2005*, volume 3672 of *Lecture Notes in Computer Science*, pages 268–283, 2005. 193

[129] Erik Poll and Aleksy Schubert. Verifying an implementation of SSH. In *Proceedings of the Workshop on Issues in the Theory of Security, WITS 2007*, pages 164–177, 2007. 186

[130] Erik Poll, Joachim van den Berg, and Bart Jacobs. Specification of the JavaCard API in JML. In *Proceedings of the 4th Working Conference on Smart Card Research and Advanced Applications*, pages 135–154. Kluwer Academic Publishers, 2001. 187

[131] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '99 (Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS '99)*, volume 1579 of *Lecture Notes in Computer Science*, pages 89–103, 1999. 190

[132] Silvio Ranise and Cesare Tinelli. *The SMT-LIB Standard: Version 1.2*, 2006. http://combination.cs.uiowa.edu/smtlib/papers/format-v1.2-r06.08.30.pdf. 18, 59

[133] Sandip Ray, John Matthews, and Mark Tuttle. Certifying compositional model checking algorithms in ACL2. In *ACL2 '03: Proceedings of the 4th International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003. 191, 197

[134] Erik Reeber and Warren A. Hunt Jr. A SAT-based decision procedure for the Subclass of Unrollable List Formulas in ACL2 (SULFA). In *Proceedings of the Third International Joint Conference on Automated Reasoning, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 453–467, 2006. 191

[135] Wolfgang Reif. The KIV-approach to software verification. In *KORSO: Methods, Languages, and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Artificial Intelligence*, pages 339–368. 1995. 192

[136] Jan Reineke. Shape analysis of sets. Master's thesis, Saarland University, 2005. 193

[137] Thomas Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *Proceedings of the 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 380–398, 2003. 193

[138] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. 194

[139] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):942–991, 1997. 201

[140] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, 2003. 197

[141] Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking JML specifications using an extensible software model checking framework. *International Journal on Software Tools for Technology Transfer*, 8(3):280–299, 2006. 16, 197

[142] Andreas Roth. Deduktiver Softwareentwurf am Beispiel des Java Collections Frameworks. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe, June 2002. 187

[143] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL '99: Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–118, 1999. 193

[144] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002. 16, 193

[145] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2005*, volume 3385 of *Lecture Notes in Computer Science*, pages 199–215, 2005. 62

[146] Raluca Sauciuc. Data structure verification using a proof-generating theorem prover, 2005. Diploma Thesis, Politehnica University of Bucharest. 196

[147] Jun Sawada and Erik Reeber. ACL2SIX: A hint used to integrate a theorem prover and an automated verification tool. In *Proceedings of the 6th International Conference on Formal Methods in Computer-Aided Design, FMCAD '06*, pages 161–170, 2006. 191

[148] Peter H. Schmitt and Isabel Tonin. Verifying the Mondex case study. In *Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods, SEFM 2007*, pages 47–56, 2007. 187

[149] Stephan Schulz. E – A brainiac theorem prover. *Journal of AI Communications*, 15(2,3):111–126, 2002. 18, 59, 169

[150] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272, 2005. 197

[151] Kurt Stenzel. A formally verified calculus for full Java Card. In *Proceedings of the 10th International Conference on Algebraic Methodology and Software Technology, AMAST 2004*, volume 3116 of *Lecture Notes in Computer Science*, pages 491–505, 2004. 192

[152] Geoff Sutcliffe and Christian Suttner. The TPTP problem library. *Journal of Automated Reasoning*, 21(2):177–203, 1998. 59

[153] Isabel Tonin. Verifying the Mondex case study: The KeY approach. Technical Report 2007-4, University of Karlsruhe, 2007. 16, 187

[154] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *IEEE/ACM International Conference on Automated Software Engineering*, 10(2):203–232, April 2003. 197

[155] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *ISSTA '04: Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 97–107, 2004. 197

[156] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1965–2013. Elsevier Science Publishers B. V., 2001. 18, 30, 59, 169

[157] Makarius Wenzel and Stefan Berghofer. *The Isabelle System Manual.* TU München, April 2009. 189

[158] Markus M. Wenzel. *Isabelle/Isar—A Versatile Environment for Human-Readable Formal Proof Documents.* PhD thesis, Institut für Informatik, Technische Universität München, 2002. 190

[159] Thomas Wies, Viktor Kuncak, Patrick Lam, Andreas Podelski, and Martin Rinard. Field constraint analysis. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2006*, pages 157–173, 2006. 18, 59, 60, 169, 193

[160] Thomas Wies, Viktor Kuncak, Karen Zee, Andreas Podelski, and Martin Rinard. Verifying complex properties using symbolic shape analysis. In *Workshop on Heap Abstraction and Verification (collocated with ETAPS)*, 2007. 16, 17, 29, 46, 193

[161] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the 20th International Conference on Computer Aided Verification, CAV 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398, 2008. 192, 194

[162] Greta Yorsh, Alexander Rabinovich, Mooly Sagiv, Antoine Meyer, and Ahmed Bouajjani. A logic of reachable patterns in linked data-structures. In *Proceedings of the 9th International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2006*, volume 3921 of *Lecture Notes in Computer Science*, pages 94–110, 2006. 196

[163] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 349–361, 2008. 45, 169, 191

[164] Karen Zee, Viktor Kuncak, and Martin Rinard. An integrated proof language for imperative programs. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–351, 2009. 69

[165] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285, 2001. 191

[166] Dengping Zhu and Hongwei Xi. Safe programming with pointers through stateful views. In *Proceedings of the 7th International Symposium on Practical Aspects of Declarative Languages, PADL 2005*, volume 3350 of *Lecture Notes in Computer Science*, pages 83–97, 2005. 16