Biblioteca Digital
FCEN-UBA

BIBLIOTECA CENTRAL LUIS F LELOIR
BIBLIOTECA CENTRAL LELOIR
FACULTAD DE CIENCIAS EXACTAS Y NATURALES UBA

Tesis Doctoral

Verificación de software usando Alloy

Galeotti, Juan Pablo

2010

Este documento forma parte de la colección de tesis doctorales y de maestría de la Biblioteca Central Dr. Luis Federico Leloir, disponible en digital.bl.fcen.uba.ar. Su utilización debe ser acompañada por la cita bibliográfica con reconocimiento de la fuente.

This document is part of the doctoral theses collection of the Central Library Dr. Luis Federico Leloir, available in digital.bl.fcen.uba.ar. It should be used accompanied by the corresponding citation acknowledging the source.

Cita tipo APA:
Galeotti, Juan Pablo. (2010). Verificación de software usando Alloy. Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires.

Cita tipo Chicago:
Galeotti, Juan Pablo. "Verificación de software usando Alloy". Facultad de Ciencias Exactas y Naturales. Universidad de Buenos Aires. 2010.

EXACTAS UBA
Facultad de Ciencias Exactas y Naturales

UBA
Universidad de Buenos Aires

Dirección: Biblioteca Central Dr. Luis F. Leloir, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires. Intendente Güiraldes 2160 - C1428EGA - Tel. (++54 +11) 4789-9293

Contacto: digital@bl.fcen.uba.ar

UNIVERSIDAD DE BUENOS AIRES
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Verificación de Software Usando Alloy

Tesis presentada para optar al título de Doctor de la Universidad de Buenos Aires en el área Ciencias de la Computación

Lic. Juan Pablo Galeotti

Director de tesis: Dr. Marcelo Fabián Frias
Consejero de estudios: Dr. Victor Braberman

Buenos Aires, Noviembre de 2010

Verificación de Software Usando Alloy

Resumen: La verificación acotada de software usando SAT consiste en la traducción del programa junto con las anotaciones provistas por el usuario a una fórmula proposicional. Luego de lo cual la fórmula es analizada en busca de violaciones a la especificación usando un SAT-solver. Si una violación es encontrada, una traza de ejecución exponiendo el error es exhibida al usuario.

Alloy es un lenguaje formal relacional que nos permite automáticamente analizar especificiaciones buscando contraejemplos de aserciones con la ayuda de un SAT-solver *off-the-shelf*.

Las contribuciones de la presente tesis son principalmente dos. Por un lado, se presenta una traducción desde software anotado en lenguaje JML al lenguaje Alloy. Para conseguir esto, se presenta:

- DynAlloy, una extensión al lenguaje de especificación Alloy para describir propiedades dinámicas de los sistemas usando acciones. Extendemos la sintaxis de Alloy con una notación para escribir aserciones de correctitud parcial. La semántica de estas aserciones es una adaptación del precondición liberal más débil de Dijsktra.

- DynJML, un lenguaje de especificación orientado a objetos que sirve de representación intermedia para facilitar la traducción de JML a DynAlloy.

- TACO, un prototipo que implementa la traducción de JML a DynAlloy.

En segundo lugar, introducimos una técnica novedosa, general y complementamente automatizable para analizar programas Java secuenciales anotados con JML usando SAT. Esta técnica es especialmente beneficiosa cuando el programa opera con estructuras de datos complejas. Para esto, se instrumenta el modelo Alloy con un predicado de ruptura de simetrías que nos permite el cómputo paralelo y automatizado de cotas ajustadas para los campos Java.

Palabras clave: Verificación, lenguajes, análisis estático, análisis de programas usando SAT, Alloy, KodKod, DynAlloy.

Software Verification using Alloy

Abstract: SAT-based bounded verification of annotated code consists of translating the code together with the annotations to a propositional formula, and analyzing the formula for specification violations using a SAT-solver. If a violation is found, an execution trace exposing the error is exhibited.

Alloy is a formal specification language that allows us to automatically analyze specifications by searching for counterexamples of assertions with the help of the off-the-shelf SAT solvers.

The contributions of this dissertation are twofold. Firstly, we present a translation from Java Modelling Language (a behavioural specification language for Java) to Alloy. In order to do so, we introduce:

- DynAlloy, an extension to the Alloy specification language to describe dynamic properties of systems using actions. We extend Alloy's syntax with a notation for partial correctness assertions, whose semantics relies on an adaptation of Dijkstra's weakest liberal precondition.
- DynJML, an intermediate object-oriented specification language to alleviate the burden of translating JML to DynAlloy.
- TACO, a prototype tool which implements the entire tool-chain.

Secondly, we introduce a novel, general and fully automated technique for the SAT-based analysis of JML-annotated Java sequential programs dealing with complex linked data structures. We instrument Alloy with a symmetry-breaking predicate that allows for the parallel, automated computation of tight bounds for Java fields. Experiments show that the translations to propositional formulas require significantly less propositional variables, leading to a speedup on the analysis of orders of magnitude compared to the non-instrumented SAT-based analysis.

Keywords: Verification, languages, static analysis, SAT-based code analysis, Alloy, KodKod, DynAlloy.

Acknowledgements

Many people deserve my thanks and gratitude for making this work possible. I apologize in advance to those who I will not explicitly mention here due to my lack of memory.

First, I would like to thank Marcelo Frias, my supervisor. He was a source of inspiration, support and an example of the kind of researcher I would like to be in the future. Second, I would like to thank my colleagues at the RFM Group: Carlos Lopez Pombo, Mariano Moscato, Nicolas Rosner, and Ignacio Vissani,and the rest of my colleagues at LAFHIS research Lab, especially Victor Braberman, Nicolas D'Ippolito, Guido De Caso, Diego Garbervetsky, and Sebastian Uchitel. Nazareno Aguirre from Universidad Nacional de Rio Cuarto was also a valuable colleague along all these years. I would also want to thank all the undergrad students I had the honour to supervise: Pablo Bendersky, Brian Cardiff, Diego Dobniewski, Gabriel Gasser Noblia and Esteban Lanzarotti. I especially want to thank Nicolas Kicillof. Due to his advice I ended up studying Computer Science.

My summer internship at Microsoft Research was a wonderful experience. I would like to thank my mentors: Shuvendu Lahiri and Shaz Qadeer. I would also want to thank Rustan Leino and Phillip Rummer for making my stay in Redmond so nice.

Several researchers helped me tremendously during all these years. I thank Karen Zee, Kuat Yessenov, Greg Dennis, Emina Torlak, Felix Chang, Willem Visser, Robby and Esteban Mocskos for their help and clarifications.

Last but not least, I thank the thesis committee members for their comments and suggestions for improving the final version of this dissertation: Daniel Jackson, Darko Marinov and Alfredo Olivero.

Dedicatoria

Dedico esta tesis de doctorado a mis amigos, a mi hermano, a mi papá y a mi mamá. A mi esposa Elena, a la que amo con todo mi corazón, y a las *globinas* que amo mucho.

Contents

1 Introduction **15**
- 1.1 Background . . . 16
  - Alloy . . . 16
- 1.2 Efficient intra-procedural SAT-based program analysis . . . 17
  - Our approach . . . 17
  - Evaluation . . . 18
- 1.3 Thesis . . . 19
- 1.4 Contributions . . . 19
- 1.5 Organization . . . 20

2 Preliminaries **22**
- 2.1 An overview of Alloy . . . 22
- 2.2 The Alloy Language . . . 25
- 2.3 Operations in Alloy . . . 27
- 2.4 Properties of a Model . . . 29
- 2.5 Alloy Assertions . . . 30

3 DynAlloy: extending Alloy with procedural actions **33**
- 3.1 Predicates vs. Actions . . . 33
- 3.2 Syntax and Semantics of DynAlloy . . . 34
- 3.3 Specifying Properties of Executions in DynAlloy . . . 35
- 3.4 Analyzing DynAlloy Specifications . . . 37

4 DynJML: a relational object-oriented language **40**
- 4.1 DynJML syntax and semantics . . . 40
  - Specifying program behaviour . . . 42
  - Abstractions . . . 44
  - Object Invariants . . . 45
  - Modifying the system state . . . 46

Memory allocation and program invocation 47
Program inheritance 48
Program overloading 50
Abstract programs 50
Program invocation in specifications 51
Assertions 51
Loop invariants 52
The Assume and Havoc Statements 53
4.2 Analyzing DynJML specifications 54
Adding signatures to DynAlloy 54
Modeling actions 55
Translating a program implementation 56
Partial Correctness Assertions 58
Procedure calls 61
Transforming Assertion statements 64
Transforming Loop invariants 65
Modular SAT-based Analysis 67

5 TACO: from JML to SAT **68**
5.1 Java Modeling Language (JML) 70
5.2 Translating JML to DynJML 74
Initial transformations 74
Translation to DynJML 75
Exceptions and JML Behaviours 77
JDK classes 78
5.3 Bounded Verification 79
5.4 Implementation Details 79

6 A New Predicate for Symmetry Breaking **81**
6.1 SAT-based symmetry breaking 81
6.2 An algorithm for generating symmetry breaking axioms 83
Symmetry breaking predicates: An example 89
6.3 A Correctness proof 96

7 Parallel Computation of Tight Bounds **99**
7.1 Symmetry Breaking and Tight Bounds 102
7.2 An iterative algorithm for bound computation 103
7.3 An eager algorithm for bound computation 105

8 Evaluation **110**
8.1 Experimental Setup . . . 110
8.2 Analyses Using Symmetry Breaking Predicates . . . 112
8.3 Computing Tight Bounds . . . 113
8.4 Analysing the Impact of Using Bounds . . . 115
8.5 Analysis of Bug-Free Code . . . 119
8.6 Bug Detection Using TACO . . . 119
Detecting Mutants . . . 120
Detecting a Seeded Non-Trivial Bug . . . 126
Detecting a Previously Unknown Fault . . . 128
8.7 Threats to Validity . . . 130
8.8 Chapter summary . . . 132

9 Related work **134**
9.1 Java SAT-based bounded verification . . . 134
9.2 C SAT-based bounded verification . . . 136
9.3 Theorem Proving . . . 138
SMT-based program verification . . . 138
Jahob . . . 139
9.4 Model checking Java programs . . . 139
9.5 Related heap canonization . . . 140
9.6 Related extensions to Alloy . . . 141
9.7 Related tight bound computation . . . 141
9.8 Shape Analysis . . . 142

10 Conclusions **143**
10.1 Conclusions . . . 143
10.2 Future work . . . 144

A DynJML Grammar **154**

List of Figures

2.1 Grammar and semantics of Alloy 31
2.2 An Alloy counterexample visualization 32

3.1 Grammar for composite actions in DynAlloy 35
3.2 Semantics of DynAlloy. 36

5.1 Translating annotated code to SAT 69

6.1 Two isomorphic list instances found by Alloy Analyzer 83
6.2 The *instrument_Alloy()* procedure 84
6.3 The *local_ordering()* procedure 86
6.4 The *global_ordering()* procedure 87
6.5 The *define_min_parent()* procedure 87
6.6 The *define_freach()* procedure 87
6.7 Comparing nodes using their min-parents. 88
6.8 The *order_root_nodes()* procedure 89
6.9 The *root_is_minimum()* procedure 90
6.10 The *order_same_min_parent()* procedure 91
6.11 The *order_same_min_parent_type()* procedure 92
6.12 The *order_diff_min_parent_types()* procedure 92
6.13 The *avoid_holes()* procedure 92
6.14 A red-black trees class hierarchy 93

7.1 Matrix representation of an Alloy field. 99
7.2 The naive parallel algorithm for bound refinement. 104
7.3 TACO's algorithm for iterative bound refinement. 107
7.4 TACO architecture extended with a bounds repository. 108
7.5 TACO's algorithm for eager bound refinement. 109

8.1 Analyses time (in logarithmic scale) for computing bounds of TreeSet using the iterative and the eager algorithms 117

8.2 Analyses time (in logarithmic scale) for computing bounds of Binomial heap using the iterative and the eager algorithms . . . 118
8.3 Analysis time as bound precision is increased. 119
8.4 Analysis time as bound precision is increased. 120
8.5 Analysis time as bound precision is increased. 121
8.6 Analysis time for method *remove* of AvlTree as scope and bound tightness grows. 122
8.7 Analysis time for method *insert* of CList as scope and bound tightness grows. 123
8.8 Efficacy of JForge, TACO$^-$ and TACO for mutants killing. . . . 125
8.9 Code snippets from `CList.remove` (a), and a bug-seeded version (b). 127
8.10 A 13 nodes heap that exhibits the failure in method `ExtractMin`. 131

List of Tables

5.1 Transforming Java expressions 75
5.2 Transforming Java conditionals 76

8.1 Comparison of code analysis times for 10 loop unrolls using TACO^{-} (T^{-}) and TACO^{sym} (T^{s}). 113
8.2 Comparison of code analysis times for 10 loop unrolls using TACO^{-} (T^{-}) and TACO^{sym} (T^{s}). 114
8.3 Sizes for initial upper bounds (#UB) and for tight upper bounds (#TUB), and analysis time for computation of tight upper bounds using iterative algorithm (I) and eager algorithm (E). 116
8.4 Comparison of code analysis times for 10 loop unrolls using JForge (JF) and TACO (T). 124
8.5 Analysis times for mutants killing. TACO times reflect the analysis time plus the bounds computation time. 125
8.6 Comparison of analysis behavior for some selected mutants. Analysis time for TACO includes the time required to compute the tight bound amortized among the mutants in each class. 126
8.7 Outcome of the analysis `maxCS` = 20. Ten hours timeout. TACO's bound computation is amortized. 128
8.8 Up to 2 unrolls and varying `maxCS`. 10 hours timeout. 128

Chapter 1

Introduction

Today's software is everywhere, from small cell phones to large information banking systems. Software is not just a marketable product anymore but an essential factor embedded in everything in modern life and a major driver of the global economy. Today's software is no longer a few hundred lines of code written to run on some computers but several hundred million lines of code that control some of the most complicated systems humans have ever created.

As software expands its presence from desktop computers to embedded components such as home appliances, personal digital assistants, medical equipment, etc., this expansion comes with an undesirable companion: software defects, most commonly known as *software bugs*. Software failures range from those that we may consider annoying [17] to those with more tragic consequences [38]. According to a study of the US National Institute of Standards & Technology (NIST), only for 2002 the annual cost of software failure represented the .6 % of the U.S. GDP [69].

The increasing importance of software quality in economy and in everyday life demands the development of (a) more robust engineering techniques and processes to build software, and (b) more advanced tools to help programmers to achieve a greater quality in the software artifacts they produce. Modern compilers benefit from program analysis techniques such as type checking and data flow analysis to warn programmers about unintentional mistakes. In both cases, the degree of automation is extremely high. Similarly, in both cases the programmer remains almost unaware of the decision procedure being applied to analyze his or her code.

Our work assumes that formal methods commonly known as "light-weight" may play a key role in devising a verification tool for improving the quality of software. It is our belief that essential requirements for a tool to become part of

a programmer toolkit are (a) its proximity to programmer's mindset and (b) its "push-button" nature.

1.1 Background

Bounded Verification [28] is a technique in which all executions of a procedure are exhaustively examined within (a) a finite space given by a bound on the size of the heap and (b) the number of loop unrollings. The scope of analysis is examined in search of an execution trace which violates a given specification.

Java Modeling Language (JML) is a behavioral specification language for Java inspired by the design-by-contract programming methodology. JML annotations allow programmers to define a method contract (using constructs such as requires, ensures, assignable, signals, etc.), and invariants (both static and non-static). A contract may include normal behavior (how does the system behave when no exception is thrown) and exceptional behavior (what is the expected behavior when an exception is thrown).

Several bounded verifications tools [15,21,27,31,52,90] rely on appropriately translating the original piece of software, as well as the specification to be verified, to a propositional formula. The use of a SAT-Solver [33,42,67] then allows one to find a valuation for the propositional variables that encodes a failure.

It is well known that the SAT problem is NP-complete [18]. Thus, the time required for solving an instance of this problem is (provided P $\neq$ NP) exponential in the amount of propositional variables of the formula resulting from the translation of source code. Although the worst-case time is exponential in general, modern SAT solvers achieve better results on practical problems

Alloy

Alloy [48,49,51] is a formal specification language, which belongs to the class of the so-called model-oriented formal methods. Alloy is defined in terms of a simple relational semantics, its syntax includes constructs ubiquitous in object-oriented notations, and it features automated analysis capabilities [47].

The Alloy language has been designed with the goal of making specifications automatically analyzable. As the aforementioned bounded verification tools, the Alloy Analyzer tool relies on off-the-shelf SAT solvers. It automatically performs an exhaustive search for counterexamples within a finite scope of analysis.

Valuations for formulas coming from Alloy specifications admit a partitioning into classes such that if a valuation in one of these classes satisfies the formula, then any other valuation of the same class also satisfies the formula because all

of them express isomorphic models of the original problem. These isomorphic models reveal what is frequently referred to as *symmetries* [77].

To cope with this problem Alloy [52] includes symmetry-breaking predicates which provide a partition into equivalence classes. Only one model for each class is evaluated, thus reducing the search space. As breaking all symmetries would require, in the general case, the construction of an exponentially large symmetry breaking predicate, Alloy only constructs a small, polynomially-sized predicate that breaks many (but often not all) symmetries. This is usually enough to get significant performance improvements.

1.2 Efficient intra-procedural SAT-based program analysis

In the presence of contracts for invoked methods, modular SAT-based analysis can be done by first replacing the calls in a method by the corresponding contracts and then analyzing the resulting code. This is the approach followed for instance in Dennis et al. [27]. One important limitation remains at the *intra-procedural* level, where the code for a single method (already including the contracts or the inlined code for called methods) has to be analyzed. Code involving linked data structures with rich invariants (such as circular lists, red-black trees, AVL trees or binomial heaps) is hard to analyze using these techniques.

Although many failures may be exposed by analyzing a small portion of the program domain (what is known as the *small scope hypothesis* [50]), improving the analysis time is fundamental for the responsible adoption of SAT-based analysis of programs.

In our opinion, SAT-based bounded verification has been perceived as an intrinsically non-scalable technique. The reason is that the translation of a complete system to a propositional formula, and the analysis of such formula using a SAT-solver, are very likely not to scale. We believe this is mostly true unless some careful decisions are made. For instance, it is worth accepting that SAT-based analysis (as described) is not meant to be a monolithic process to be applied to large pieces of software. Also, it is important to understand the reasons for the non-scalability of SAT-solving, and act in order to minimize their impact during analysis. Finally, it is essential to fully understand what are the benefits of SAT-based analysis when compared to other analysis techniques.

Our approach

From the technical point of view, we present a novel, general and fully automated technique for the intra-procedural analysis of JML-annotated Java code.

The presented technique is consistent with the methodology hereby presented. To improve the analysis time, we can then proceed in two ways:

- Reducing the space of valuations to be considered by the SAT-solver. Our intention is to eliminate those valuations that may not lead to feasible counterexamples.
- Reducing the number of propositional variables in the propositional formula result of the translation.

We have developed a technique that combines both approaches in a synergistic and fully automated way. First, it forces a canonical representation of the Java memory heap by removing symmetries of object references. This greatly reduces the number of meaningful valuations to be considered by the SAT-solver. Second, and as a consequence of the heap canonicalization, a simple preprocessing makes possible to determine in advance the truth value of a substantial proportion of the propositional variables. These variables can be replaced by their predetermined truth value, yielding a simpler SAT problem.

We have implemented this technique in a bounded verification tool named TACO (after Translation of Annotated COde). TACO translates JML-annotated Java code to a SAT problem, using Alloy as an intermediate language.

Evaluation

Evaluations of techniques for bounded verification center around two aspects: (1) the ability to exhaust the search space when there is no violation to the specification, and (2) the performance for finding bugs when the implementation is faulty.

As code involving linked data structures with rich invariants is hard to analyze using bounded verification techniques, we define a benchmark of several collection classes with increasingly complex class invariants. As we will see, these structures have become accepted benchmarks for comparison of analysis tools in the program analysis community (see for instance [11, 27, 52, 76, 86]).

Once the benchmark for the evaluation is defined, we ask the following questions:

- Are the new symmetry breaking predicates we propose amenable to the SAT-Solving process?
- Is the analysis sensitive to removing propositional variables from infeasible initial states?

- How does this technique compare to current state-of-the-art tools based on SMT-Solving and model-checking?

As we will present in Chapter 8, the experiments allow us to conclude that the inclusion of the symmetry-breaking predicates by itself increases scalability: 94% of cases show an increase of the maximum scope of analysis of 7.31 nodes, on average. As a hint of the power of the symmetry breaking predicates, it allowed us to reduce the analysis time of a method for finding an element in an AVL tree from over 10 hours to 87 seconds.

We also conclude that the SAT-Solving analysis is sensitive to tightening the upper bounds: 85% of experiments do exhibit an exponential improvement as the number of propositional variables from the initial state are removed. For the same AVL tree experiment we reported in the previous paragraph, after removing propositional variables that TACO deemed unnecessary, the analysis time reduced to 1 second.

Finally, we compare TACO against JForge [28] (a state-of-the-art SAT-based analysis tool developed at MIT), ESC/Java2 [13], Java PathFinder [85], and Kiasan [9]. TACO outperforms JForge when analyzing bug-free as well as faulty software (i.e., mutant-generated and manually seeded programs). TACO also outperforms in these experiments Java PathFinder, Kiasan and ESC/Java2. For the particular task of finding a previously unknown bug requiring 13 node elements in the initial heap and exercising at most 4 loop unrolls, TACO succeeded in finding an offending trace in 53 seconds. JForge, and Java PathFinder reached the 1 hour time limit. On the other hand, Kiasan exhausted the RAM memory. This provides evidence of the efficacy of the proposed approach for improving SAT-based verification of complex linked-data structures.

1.3 Thesis

Our thesis is twofold. First, we sustain that Alloy is a suitable target for translating sequential JML-annotated Java code using Dijkstra's weakest precondition [30]. Second, symmetry breaking and unnecessary propositional variable elimination improves several orders of magnitude the analysis time of a SAT-based bounded verification tool. This dissertation presents the evidence that supports our thesis.

1.4 Contributions

The contributions of this dissertation are:

- We add to Alloy the possibility of defining actions and asserting properties using partial correctness assertions, as a mechanism for the specification of operations. We refer to this extension of Alloy as DynAlloy.
- We present a modification of the Alloy tool in order to allow for an efficient verification of DynAlloy specifications.
- We present an intermediate representation between DynAlloy and JML called DynJML. We show how to perform a semantically preserving translation from JML specifications to DynJML language.
- We present a novel and fully automated technique for canonicalization of the memory heap in the context of SAT-solving, which assigns identifiers to heap objects in a well-defined manner (to be made precise in Chapter 6).
- Using this ordering, we present a fully automated and parallel technique for determining which variables can be removed. The technique consists of computing bounds for Java fields (to be defined in Chapter 7). The algorithm only depends on the invariant of the class under analysis. Therefore, the computed bounds can be reused across all the analysis for a class, and the cost of computing the bounds can be amortized.
- We present several case studies using complex linked data structures showing that the technique improves the analysis by reducing analysis times up to several orders of magnitude in case correct code is analyzed. We also show that the technique can efficiently discover errors seeded using mutant generation [24]. Finally, we report on a previously unknown [87] error found in a benchmark used for test-input generation [86]. This error was not detected by several state-of-the-art tools based on SAT-solving, model checking or SMT-solving.
- We discuss to what extent the techniques presented in this dissertation can be used by related tools.

1.5 Organization

The remainder of this dissertation is organized as follows. In Chapter 2 we present in more detail the fundamental concepts required to follow the work presented in this dissertation. In Chapter 3 we present our extension to Alloy with procedural actions, DynAlloy. In Chapter 4 we show our intermediate representation between DynAlloy and JML. In Chapter 5 we present TACO as a

tool to perform bounded verification of software. In Chapter 6 we present a new symmetry-breaking predicate amenable to SAT-Solving. In Chapter 7 we present a novel and fully automatic technique for removing unnecessary propositional variables. In Chapter 8 we report extensive experimental results on the scalability of the proposed technique. In Chapter 9 we discuss some relevant related work. Finally, in Chapter 10 we present our conclusions and some future research lines.

Chapter 2

Preliminaries

This chapter defines and sets notation for the concepts that will be needed to understand the rest of thesis. It does not aim to be a complete presentation of the topic. Instead it showcases the most significant notions using formal definitions when required for a better understanding, but resorting to informal explanations (and corresponding citations for a full presentation) if possible.

2.1 An overview of Alloy

Alloy [48,49,51] is a formal specification language, which belongs to the class of the so-called model-oriented formal methods. Alloy is defined in terms of a simple relational semantics, its syntax includes constructs ubiquitous in object-oriented notations, and it features automated analysis capabilities [47]; these characteristics have made Alloy an appealing formal method.

Alloy has its roots in the *Z* specification language [79], and, as *Z*, is appropriate for describing structural properties of systems. However, in contrast with *Z*, Alloy has been designed with the goal of making specifications automatically analyzable.

Alloy's representations of systems are based on abstract models. These models are defined essentially in terms of *data domains*, and *operations* between these domains. In particular, one can use data domains to specify the state space of a system or a component, and employ operations as a means for the specification of state change. Semantically, operations correspond to *predicates*, in which certain variables are assumed to be *output* variables, or, more precisely, are meant to describe the system state *after* the operation is executed. By looking into Alloy's semantics, it is easy to verify that "output" and "after" are *intentional concepts*, i.e., the notions of output or temporal precedence are not reflected in

the semantics and, therefore, understanding variables this way is just a (reasonable) convention. Variable naming conventions are a useful mechanism, which might lead to a simpler semantics of specifications. However, as we advocate in this paper, the inclusion of *actions* (understood as a general concept associated with state change, covering transactions and events, for example), with a well defined input/output semantics, in order to specify properties of executions, might provide a significant improvement to Alloy's expressiveness and analyzability. Moreover, actions enable us to characterise properties regarding execution traces in a convenient way.

In order to see how actions might improve Alloy's expressiveness, suppose, for instance, that we need to define the combination of certain operations describing a system. Some combinations are representable in Alloy; for instance, if we have two operations $Oper_1$ and $Oper_2$, and denote by $Oper_1 ; Oper_2$ and $Oper_1 + Oper_2$ the sequential composition and nondeterministic choice of these operations, respectively, then these can be easily defined in Alloy as follows:

$$Oper_1 ; Oper_2(x, y) = \text{some } z \mid (Oper_1(x, z) \text{ and } Oper_2(z, y)),$$

$$Oper_1 + Oper_2(x, y) = Oper_1(x, y) \text{ or } Oper_2(x, y) \ .$$

However, if we aim at specifying properties of executions, then it is reasonable to think that we will need to predicate at least about all terminating executions of the system. This demands some kind of iteration of operations. While it is possible to define sequential composition or nondeterministic choice, as we showed before, finite (unbounded) iteration of operations cannot be defined in Alloy.

Nevertheless, some effort has been put toward representing the iteration of operations, in order to analyze properties of executions in Alloy. By enriching models with the inclusion of a new signature (type) for execution traces [51], and constraints that indicate how these traces are constructed from the operations of the system, it is possible to *simulate* operation iteration. Essentially, traces are defined as being composed of all intermediate states visited along specific runs. While adding traces to specifications provides indeed a mechanism for dealing with executions (and even specifications involving execution traces can be automatically analyzed), this approach requires the specifier to explicitly take care of the definition of traces (an *ad hoc* task which depends on the properties of traces one wants to validate). Furthermore, the resulting specifications are cumbersome, since they mix together two clearly separated aspects of systems, the *static* definition of domains and operations that constitute the system, and the *dynamic* specification of traces of executions of these operations. We consider

that actions, if appropriately used, constitute a better candidate for specifying assertions regarding the dynamics of a system (i.e., assertions regarding execution traces), leading to cleaner specifications, with clearer separation of concerns.

In order to compare these two approaches, let us suppose that we need to specify that every terminating arbitrary execution of two operations $Oper_1$ and $Oper_2$ beginning in a state satisfying a formula α terminates in a state satisfying a formula β. Using the approach presented in Jackson et al. [51], it is necessary to provide an explicit specification of execution traces complementing the specification of the system, as follows:

1. specify the initial state as a state satisfying α,
2. specify that every pair of consecutive states in a trace is either related by $Oper_1$ or by $Oper_2$,
3. specify that the final state satisfies β.

Using the approach we propose, based on actions, execution traces are only implicitly used. The above specification can be written in a simple and elegant way, as follows:

$$\begin{gathered} \{\alpha\} \\ (Oper_1 + Oper_2)^* \\ \{\beta\} \end{gathered}$$

This states, as we required, that every terminating execution of $(Oper_1 + Oper_2)^*$ (which represents an unbounded iteration of the nondeterministic choice between $Oper_1$ and $Oper_2$) starting in a state satisfying α, ends up in a state satisfying β. This notation corresponds to the traditional and well-known notation for partial correctness assertions. Notice that no explicit reference to traces is required. Nevertheless, traces exist and are well taken care of in the semantics of actions, far from the eyes of the software engineer writing a model. It seems clear then that pursuing our task of adding actions to Alloy might indeed contribute toward the usability of the language. Note that finite unbounded iteration is, in our approach, expressible via the iteration operation "*".

As we mentioned, one of the main features of Alloy is its analyzability. The Alloy tool allows us to automatically analyze specifications by searching for counterexamples of assertions with the help of the off-the-shelf SAT solvers MiniSAT [33], ZChaff [67] and Berkmin [42]. Therefore, extending the language with actions, while still an interesting intellectual puzzle, is not important if it cannot be complemented with efficient automatic analysis. So, we modify the

Alloy tool in order to deal with the analysis of Alloy specifications involving actions and execution traces assertions. Notice that, even though finite unbounded iteration is expressible in DynAlloy, a bound on the depth of the iterations needs to be imposed for the analysis tasks. So, for SAT solving based analysis, our extension only covers bounded iteration.

2.2 The Alloy Language

In this section, we introduce the reader to the Alloy specification language by means of an example inspired from [49]. This example serves as a means for illustrating the standard features of the language and their associated semantics, and will also help us demonstrate the shortcomings we wish to overcome.

Suppose we want to specify a simple address book for an email client. We might recognize that, in order to specify address books, data types for names and addresses are especially necessary. We can then start by indicating the existence of disjoint sets (of atoms) for names and addresses, which in Alloy are specified using *signatures*:

sig *Address* { } sig *Name* { }

These are basic signatures. We do not assume any special properties regarding the structures of names and addresses.

With names and addresses already defined, we can now specify what constitutes an address book. A possible way of defining address books is by saying that an address book consists of set of contacts, and a (total) mapping from these names to addresses:

```
sig Book {
    contacts: set Name,
    addressOf: contacts ->one Address
}
```

The keyword "one" in the above definition indicates that "addressOf" is functional and total (for each element a of contacts, there exists exactly one element d in *Address* such that addressOf(a) = d).

Alloy allows for the definition of signatures as subsets of the set denoted by another "parent" signature. This is done via what is called *signature extension*. For the example, one could define other (perhaps more complex) kinds of address books as extensions of the *Book* signature:

```
sig SafeBook extends Book {}

sig FilteredBook extends Book {
    spammers: set Address
}
```

As specified in these definitions, *SafeBook* and *FilteredBook* are special kinds of address books. If unsolicited bulk messages are sent indiscriminately from an address it may be considered a *spammer*. A *safe* address book is intended to store all entries to whom no spam detection is required. On the other hand, messages from addresses stored at a *filtered* address book may be block by the detection software.

A system might now be defined to be composed of a safe address book and a filtered address book:

```
sig System {
    filtered: FilteredBook,
    safe: SafeBook
}
```

As the previous definitions show, signatures are used to define data domains and their structure. The attributes of a signature denote *relations*. For instance, the "contacts" attribute in signature *Book* represents a binary relation, from book atoms to sets of atoms from *Name*. Given a set bs (not necessarily a singleton) of *Book* atoms, bs.contacts denotes the relational image of bs under the relation denoted by contacts. This leads to a relational view of the dot notation, which is simple and elegant, and preserves the intuitive navigational reading of dot, as in object orientation. Signature extension, as we mentioned before, is interpreted as inclusion of the set of atoms of the extending signature into the set of atoms of the extended signature.

In Fig. 2.1, we present the grammar and semantics of Alloy's relational logic, the core logic on top of which all of Alloy's syntax and semantics are defined. An important difference with respect to previous versions of Alloy, as the one presented in Jackson [48], is that expressions now range over relations of arbitrary rank, instead of being restricted to binary relations. Composition of binary relations is well understood; but for relations of higher rank, the following definition for the composition of relations has to be considered:

$$R;S = \{\langle a_1, \ldots, a_{i-1}, b_2, \ldots, b_j\rangle : \exists b \left(\langle a_1, \ldots, a_{i-1}, b\rangle \in R \ \wedge \ \langle b, b_2, \ldots, b_j\rangle \in S\right)\} .$$

Operations for transitive closure and transposition are only defined for binary relations. Thus, function X in Fig. 2.1 is partial.

2.3 Operations in Alloy

So far, we have just shown how the structure of data domains can be specified in Alloy. Of course, one would like to be able to define operations over the defined domains. Following the style of Z specifications, operations in Alloy can be defined as expressions, relating states from the state spaces described by the signature definitions. Primed variables are used to denote the resulting values, although this is just a convention, not reflected in the semantics.

In order to illustrate the definition of operations in Alloy, consider, for instance, an operation that specifies what happens when an entry is added to an address book:

```
pred AddContact[b, b': Book, n: Name, a: Address] {
    b'.contacts = b.contacts + n
    b'.addressOf = b.addressOf ++ (n -> a)
}
```
(2.1)

The intended meaning of this definition can be easily understood, having in mind that b' is meant to denote the address book (or address book state) resulting of the application of function AddContact, n -> a denotes the ordered pair ⟨n, a⟩, and ++ denotes relational overriding, defined as follows[1]:

$$R{+}{+}S \quad = \quad \{\langle a_1, \ldots, a_n\rangle : \langle a_1, \ldots, a_n\rangle \in R \ \wedge \ a_1 \notin \mathsf{dom}(S)\} \ \cup \ S \quad .$$

Other operation in an address book may include deleting a contact. In this operation $n- > Address$ denotes all the ordered pairs whose domains fall into the set n, and that range over the domain Address.

[1] Given a n-ary relation R, $\mathsf{dom}(R)$ denotes the set $\{a_1 : \exists a_2, \ldots, a_n$ such that $\langle a_1, a_2, \ldots, a_n\rangle \in R\}$.

```
pred DelContact[b, b': Book, n: Name] {
    b'.contacts = b.contacts - n
    b'.addressOf = b.addressOf - (n -> Address)
}
```
(2.2)

We have already seen a number of constructs available in Alloy, such as the dot notation and signature extension, that resemble object-oriented definitions. Operations, however, represented by functions in Alloy, are not "attached" to signature definitions, as in traditional object-oriented approaches. Instead, functions describe operations of the whole set of signatures, i.e., the model. So, there is no notion similar to that of class, as a mechanism for encapsulating data (attributes or fields) and behavior (operations or methods).

In order to illustrate a couple of further points, consider the following more complex predicate definition:

```
pred SysAdd[s, s': System] {
  some n: Name, a:Address |
    AddContact[s.filtered, s'.filtered, n, a]
    s'.filtered.spammers = s.filtered.spammers
    s'.safe = s.safe
}
```

There are two important issues exhibited in this predicate definition. First, predicate SysAdd is defined in terms of the more primitive AddContact. Second, the use of AddContact takes advantage of the *hierarchy* defined by signature extension: note that predicate AddContact was defined for address books, and in SysAdd it is being "applied" to filtered address books.

Besides adding a contact, it is necessary to model that a user may decide to stop filtering some contacts. A (nondeterministic) operation that moves contacts from the filtered book to the safe list can be specified in the following way:

```
pred SysRemoveFilter[s, s': System] {
   some n: s.filtered.contacts | {
      DelContact[s.filtered, s'.filtered, n]
      s'.filtered.spammers = s'.filtered.spammers
      AddContact[s.safe, s'.safe, n, s.addressOf[n]]
   }
}
```

The definition of SysResumeFiltering follows analogously. Finally, an operation to mark a given contact as a spammer is necessary in order to have a realistic model of address books.

```
pred SysMarkSpammer[s, s': System] {
    some n: s.filtered.contacts, a: s.filtered.addressOf[n] | {
        DelContact[s.filtered, s'.filtered]
        s'.filtered.spammers = s'.filtered.spammers + a
        s'.safe = s.safe
    }
}
```

Predicates can also be used to represent *special* states. For instance, we can characterize the states in which no spammer address is present in either the safe list nor the filtered book.

```
pred BookInv[s: System] {
    no s.filtered.spammers & Name.(s.safe.addressOf + s.filtered.addressOf)
}
```
(2.3)

In the above expression, "no x" indicates that x has no elements, and & denotes intersection.

2.4 Properties of a Model

As the reader might expect, a model can be enhanced by adding properties (axioms) to it. These properties are written as logical formulas, much in the style of the Object Constraint Language (OCL). Properties or constraints in Alloy are defined as *facts*. To give an idea of how constraints or properties are specified, we reproduce one here. It might be necessary to say that, in every system, no contact is simultaneously present in both the safe list and the filtered address book:

```
fact {
  all s: System | {
    no n : s.safe.contacts + s.filtered.contacts | {
      n in s.safe.contacts
      n in s.filtered.contacts
    }
  }
}
```
(2.4)

More complex facts can be expressed by using the quite considerable expressive power of the relational logic.

2.5 Alloy Assertions

Assertions are the *intended* properties of a given model. Consider, for instance, the following simple Alloy assertion, regarding the presented example:

```
assert {
    all s,s': System | BookInv[s] && SysAdd[s,s']
      => BookInv[s']
}
```

This assertion states that, if "BookInv" holds in system s and an addition from system s results in a new system (namely s'), then "BookInv" still holds in the resulting system .

Assertions are used to check specifications. Using the Alloy analyzer, it is possible to validate assertions, by searching for possible (finite) counterexamples for them, under the constraints imposed in the specification of the system.

Given the specification we have outlined, the user selects a scope of analysis of 3 atoms for each signature defined (namely, *Name* :, *System*, *Address* and *Book*). By combining the specification, the scope and the assertion, the Analyser builds a propositional formula stating the validity of the assertion. Since a valuation to the SAT problem is found, the assertion does not hold for the given scope. The counterexample shown in Figure 2.2 is provided to the user by interpreting the SAT valuation as a solution to the original problem.

By studying Figure 2.2 the user realizes that the assertion does not hold since the *SysAdd* operation allows an unexpected behaviour. The current specification for *SysAdd* allows adding an address (named as *SysAdd_a* in the counterexample visualization) that is contained in the set of spammer addresses of the *FilteredBook*. The user then refines the specification for the *SysAdd* operation by stating that the address is not already contained in the set of spammer addresses nor the safe book:

```
pred SysAdd[s, s': System] {
  some n: Name, a:Address - s.filtered.spammers
       - Name.(s.safe.addressOf ) - Name.(s.filtered.addressOf ) |
    AddContact[s.filtered, s'.filtered, n, a]
    s'.filtered.spammers = s.filtered.spammers
    s'.safe = s.safe
}
```

problem ::= decl*form
decl ::= *var* : *typexpr*
typexpr ::=
type
| *type* → *type*
| *type* ⇒ *typexpr*

form ::=
expr *in* expr (subset)
|!form (neg)
| form && form (conj)
| form || form (disj)
| *all v* : *type*/form (univ)
| *some v* : *type*/form (exist)

expr ::=
expr + expr (union)
| expr & expr (intersection)
| expr − expr (difference)
|~ expr (transpose)
| expr.expr (navigation)
| +expr (transitive closure)
| {*v* : *t*/form} (set former)
| *Var*

Var ::=
var (variable)
| *Var*[*var*] (application)

M : form → *env* → *Boolean*
X : expr → *env* → *value*
env = (*var* + *type*) → *value*
value = (*atom* × ⋯ × *atom*)+
  (*atom* → *value*)

$M[a\ in\ b]e = X[a]e \subseteq X[b]e$
$M[!F]e = \neg M[F]e$
$M[F\&\&G]e = M[F]e \wedge M[G]e$
$M[F \parallel G]e = M[F]e \vee M[G]e$
$M[all\ v : t/F] =$
  $\bigwedge\{M[F](e \oplus v{\mapsto}\{\,x\,\})/x \in e(t)\}$
$M[some\ v : t/F] =$
  $\bigvee\{M[F](e \oplus v{\mapsto}\{\,x\,\})/x \in e(t)\}$

$X[a + b]e = X[a]e \cup X[b]e$
$X[a\&b]e = X[a]e \cap X[b]e$
$X[a - b]e = X[a]e \setminus X[b]e$
$X[\sim a]e = \{\,\langle x, y\rangle : \langle y, x\rangle \in X[a]e\,\}$
$X[a.b]e = X[a]e;X[b]e$
$X[+a]e =$ the smallest r such that
  $r;r \subseteq r$ and $X[a]e \subseteq r$
$X[\{v : t/F\}]e =$
  $\{x \in e(t)/M[F](e \oplus v{\mapsto}\{\,x\,\})\}$
$X[v]e = e(v)$
$X[a[v]]e = \{\langle y_1, \ldots, y_n\rangle/$
  $\exists x.\, \langle x, y_1, \ldots, y_n\rangle \in e(a) \wedge \langle x\rangle \in e(v)\}$

Figure 2.1. Grammar and semantics of Alloy

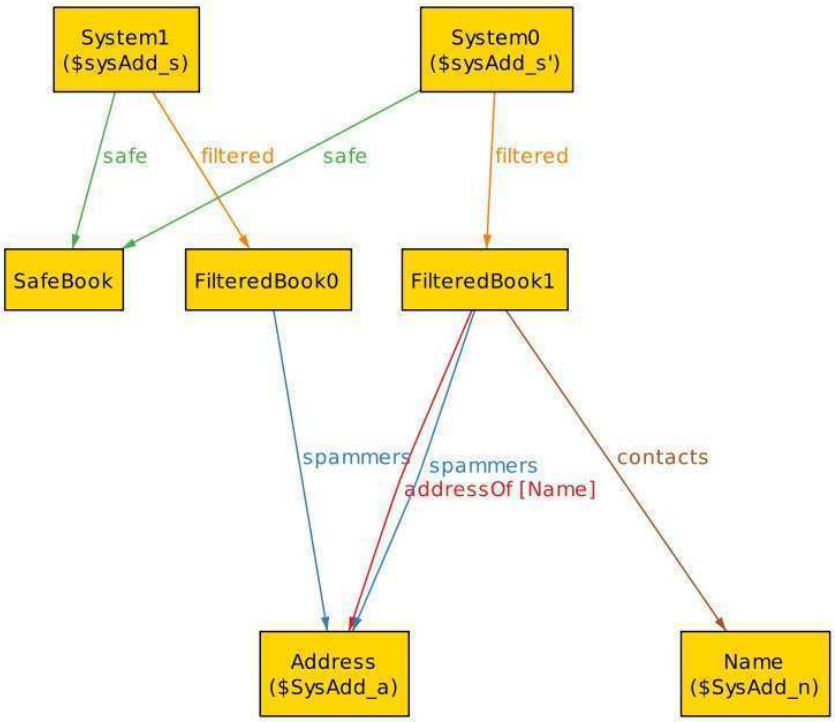


Figure 2.2. An Alloy counterexample visualization

Chapter 3

DynAlloy: extending Alloy with procedural actions

In this section we extend Alloy's relational logic syntax and semantics with the aim of dealing with properties of executions of operations specified in Alloy. It will follow that DynAlloy extends Alloy and its relational logic.

The reason for this extension is that we want to provide a setting in which, besides functions describing sets of states, actions are made available, to represent state changes (i.e., to describe relations between input and output data). As opposed to the use of predicates for this purpose, actions have an input/output meaning reflected in the semantics, and can be composed to form more complex actions, using well-known constructs from imperative programming languages.

The syntax and semantics of DynAlloy is described in Section 3.1. It is worth mentioning at this point that both were strongly motivated by dynamic logic [43], and the suitability of dynamic logic for expressing partial correctness assertions.

3.1 Predicates vs. Actions

Predicates in Alloy are just parameterized formulas. Some of the parameters are considered input parameters, and the relationship between input and output parameters relies on the convention that the second argument is the result of the operation application. Recalling the definition of predicate AddContact, notice that there is no actual change in the state of the system, since no variable actually changes its value.

Dynamic logic [43] arose in the early '70s, with the intention of faithfully reflecting state change. Motivated by dynamic logic, we propose the use of actions to model state change in Alloy, as described below.

What we would like to say about an action is how it transforms the system state after its execution. A (now) traditional way of doing so is by using pre and post condition assertions. An assertion of the form

$$\begin{array}{c} \{\alpha\} \\ A \\ \{\beta\} \end{array}$$

affirms that whenever action A is executed on a state satisfying α, if it terminates, it does so in a state satisfying β. This approach is particularly appropriate, since behaviors described by Alloy predicates are better viewed as the result of performing an action on an input state. Thus, the definition of predicate AddContact could be expressed as an action definition, of the following form:

$$\begin{array}{c} \{true\} \\ AddContact[\mathrm{b} : Book, \mathrm{n} : Name, \mathrm{a} : Address] \\ \{\mathrm{b}'.\mathrm{contacts} = \mathrm{b}.\mathrm{contacts} + \mathrm{n} \;\&\& \\ \mathrm{b}'.\mathrm{addressOf} = \mathrm{b}.\mathrm{addressOf} \mathbin{++} (\mathrm{n} \rightarrow \mathrm{a})\} \; . \end{array} \tag{3.1}$$

At first glance it is difficult to see the differences between (2.1) and (4.2), since both formulas seem to provide the same information. The crucial differences are reflected in the semantics, as well as in the fact that actions can be sequentially composed, iterated or composed by nondeterministic choice, while Alloy predicates, in principle, cannot.

An immediately apparent difference between (2.1) and (4.2) is that action *AddContact* does not involve the parameter b′, while predicate AddContact uses it. This is so because we use the convention that b′ denotes the state of variable b *after* execution of action *AddContact*. This time, "*after*" means that b′ gets its value in an environment reachable through the execution of action *AddContact* (cf. Fig. 3.2). Since *AddContact* denotes a binary relation on the set of environments, there is a precise notion of input/output inducing a before/after relationship.

3.2 Syntax and Semantics of DynAlloy

The syntax of DynAlloy's formulas extends the one presented in Fig. 2.1 with the addition of the following clause for building partial correctness statements:

$$\begin{array}{rcl} formula & ::= & \ldots \mid \{formula\}\ program\ \{formula\} \\ & & \text{"partial correctness"} \end{array}$$

The syntax for programs (cf. Fig. 3.1) is the class of regular programs defined in Harel et al. [43], plus a new rule to allow for the construction of atomic actions from their pre and post conditions. In the definition of atomic actions, $\overline{x}$ denotes a sequence of formal parameters. Thus, it is to be expected that the precondition is a formula whose free variables are within $\overline{x}$, while postcondition variables might also include primed versions of the formal parameters.

$program$	::=	$\langle formula, formula\rangle(\overline{x})$	"atomic action"
	|	$formula?$	"test"
	|	$program + program$	"non-deterministic choice"
	|	$program; program$	"sequential composition"
	|	$program^*$	"iteration"
	|	$\langle program\rangle(\overline{x})$	"invoke program"

Figure 3.1. Grammar for composite actions in DynAlloy

In Fig. 3.2 we extend the definition of function M to partial correctness assertions and define the denotational semantics of programs as binary relations over *env*. The definition of function M on a partial correctness assertion makes clear that we are actually considering a partial correctness semantics. This follows from the fact that we are not requesting environment e to belong to the domain of the relation $P[p]$. In order to provide semantics for atomic actions, we will assume that there is a function A assigning, to each atomic action, a binary relation on the environments. We define function A as follows:

$$A(\langle pre, post\rangle) \quad = \quad \{\, \langle e, e'\rangle : M[pre]e \wedge M[post]e' \,\} \ .$$

There is a subtle point in the definition of the semantics of atomic programs. While actions may modify the value of all variables, we assume that those variables whose primed versions do not occur in the post condition retain their input value. Thus, the atomic action *AddContact* modifies the value of variable b, but a and d keep their initial values. This allows us to use simpler formulas in pre and post conditions.

3.3 Specifying Properties of Executions in DynAlloy

Suppose we want to specify that a given property P is invariant under sequences of applications of the previously defined operations ("SysAddContact", "SysRemoveFilter" and "SysResumeFiltering"), from a certain initial state. A

$$M[\{\alpha\}p\{\beta\}]e \quad = \quad M[\alpha]e \implies \forall e' \, (\langle e, e' \rangle \in P[p] \implies M[\beta]e')$$

$$\begin{aligned}
&P : program \to \mathcal{P}(env \times env) \\
&P[\langle pre, post \rangle] = A(\langle pre, post \rangle) \\
&P[\alpha?] = \{ \langle e, e' \rangle : M[\alpha]e \wedge e = e' \} \\
&P[p_1 + p_2] = P[p_1] \cup P[p_2] \\
&P[p_1 ; p_2] = P[p_1] ; P[p_2] \\
&P[p^*] = P[p]^*
\end{aligned}$$

Figure 3.2. Semantics of DynAlloy.

technique useful for stating the invariance of a property P consists of specifying that P holds in the initial states, and that for every non initial state and every operation $O \in \{SysAddContact, SysRemvoeFilter, SysResumeFiltering\}$, the following holds:

$$P(s) \wedge O(s, s') \;\Rightarrow\; P(s') \; .$$

This specification is sound but incomplete, since the invariance may be violated in unreachable states. Of course it would be desirable to have a specification in which the states under consideration were exactly the reachable ones. As explained in Jackson et al. [51], one way to achieve this is by defining Alloy *traces* in our model. Subsequently, authors introduced Imperative Alloy [70].

Recall the specification of action SysAddContact, the specification of actions SysRemoveFilter and SysResumeFiltering is done as follows:

{*true*}

SysRemoveFilter[s : *System*]

{ some *n*:s.filtered.contacts | {
    s′.filtered.contacts = s.filtered.contacts − *n*
    s′.filtered.addressOf = s.filtered.addressOf − (*n* → *Address*)
    s′.safe.contacts = s.safe.contacts + *n*
    s′.safe.addressOf = s.safe.addressOf + +(*n* → s.filtered.addressOf[*n*])
    s′.filtered.spammers = s.filtered.spammers
}}

$\{true\}$
SysResumeFiltering[s : *System*]
{ some n:s.safe.contacts | {
s′.safe.contacts = s.safe.contacts $- n$
s′.safe.addressOf = s.safe.addressOf $-(n \rightarrow Address)$
s′.filtered.contacts = s.filtered.contacts $+ n$
s′.filtered.addressOf = s.filtered.addressOf $+ +(n \rightarrow$ s.safe.addressOf[n])
s′.filtered.spammers = s.filtered.spammers
}}

Notice that by using partial correctness statements on the set of regular programs generated by the set of atomic actions { SysAddContact, SysRemoveFilter, SysResumeFiltering }, we can assert the invariance of a property P under finite applications of actions SysAddContact, SysRemoveFilter and SysResumeFiltering in a simple and elegant way, as follows:

$$\{Init(s) \wedge P(s)\}$$
$$(\text{SysAddContact}(s) + \text{SysResumeFiltering}(s) + \text{SysStopFilter}(s))^*$$
$$\{P(s')\}$$

More generally, suppose now that we want to show that a property Q is invariant under sequences of applications of arbitrary operations $O_1, \ldots, O_k$, starting from states s described by a formula *Init*. The specification of this assertion in our setting is done via the following formula:

$$\{Init(\overline{x}) \wedge Q(\overline{x})\}$$
$$(O_1(\overline{x}) + \cdots + O_k(\overline{x}))^* \tag{3.2}$$
$$\{Q(\overline{x'})\}$$

Notice that there is no need to mention traces in the specification of the previous properties. This is because finite traces get determined by the semantics of reflexive-transitive closure.

3.4 Analyzing DynAlloy Specifications

Alloy's design was deeply influenced by the intention of producing an automatically analyzable language. While DynAlloy is, to our understanding, better suited than Alloy for the specification of properties of executions, the use of

ticks and traces as defined in Jackson et al. [51] has as an advantage that it allows one to automatically analyze properties of executions. Therefore, an almost mandatory question is whether DynAlloy specifications can be automatically analyzed, and if so, how efficiently. The main rationale behind our technique is the translation of partial correctness assertions to first-order Alloy formulas, using weakest liberal preconditions [30]. The generated Alloy formulas, which may be large and quite difficult to understand, are not visible to the end user, who only accesses the declarative DynAlloy specification.

We define below a function

$$wlp : program \times formula \rightarrow formula$$

that computes the weakest liberal precondition of a formula according to a program (composite action). We will in general use names $x_1, x_2 \ldots$ for program variables, and will use names $x'_1, x'_2, \ldots$ for the value of program variables *after* action execution. We will denote by $\alpha|_x^v$ the substitution of all free occurrences of variable x by the fresh variable v in formula α.

When an atomic action a specified as $\langle pre, post\rangle(\overline{x})$ is used in a composite action, formal parameters are substituted by actual parameters. Since we assume all variables are input/output variables, actual parameters are variables, let us say, $\overline{y}$. In this situation, function *wlp* is defined as follows:

$$wlp[a(\overline{y}), f] = \quad pre|_{\overline{x}}^{\overline{y'}} \implies \text{all } \overline{n}\left(post|_{\overline{x'}}^{\overline{n}}|_{\overline{x}}^{\overline{y'}} \implies f|_{\overline{y'}}^{\overline{n}}\right) \tag{3.3}$$

A few points need to be explained about (3.3). First, we assume that free variables in f are amongst $\overline{y'}$, $\overline{x_0}$. Variables in $\overline{x_0}$ are generated by translation *pcat* given in (3.5). Second, $\overline{n}$ is an array of new variables, one for each variable modified by the action. Last, notice that the resulting formula has again its free variables amongst $\overline{y'}$, $\overline{x_0}$. This is also preserved in the remaining cases in the definition of function *wlp*.

For the remaining action constructs, the definition of function *wlp* is the following:

$$\begin{array}{lcl} wlp[g?, f] & = & g \implies f \\ wlp[p_1 + p_2, f] & = & wlp[p_1, f] \wedge wlp[p_2, f] \\ wlp[p_1; p_2, f] & = & wlp[p_1, wlp[p_2, f]] \\ wlp[p^*, f] & = & \bigwedge_{i=0}^{\infty} wlp[p^i, f]\,. \end{array}$$

Notice that *wlp* yields Alloy formulas in all these cases, except for the iteration construct, where the resulting formula may be infinitary. In order to obtain an Alloy formula, we can impose a bound on the depth of iterations. This is equivalent to fixing a maximum length for traces. A function *Bwlp* (bounded weakest

liberal precondition) is then defined exactly as *wlp*, except for iteration, where it is defined by:

$$Bwlp[p^*, f] = \bigwedge_{i=0}^{n} Bwlp[p^i, f] \,. \tag{3.4}$$

In (3.4), n is the scope set for the depth of iteration.

We now define a function *pcat* that translates partial correctness assertions to Alloy formulas. For a partial correctness assertion $\{\alpha(\overline{y})\}\ P(\overline{y})\ \{\beta(\overline{y}, \overline{y'})\}$

$$pcat\,(\{\alpha\}\ P\ \{\beta\}) = \qquad \forall \overline{y} \left(\alpha \implies \left(Bwlp\left[p, \beta|^{\overline{x_0}}_{\overline{y}}\right]\right)|^{\overline{y}}_{\overline{y'}}|^{\overline{y}}_{\overline{x_0}} \right) \,. \tag{3.5}$$

Of course this analysis method where iteration is restricted to a fixed depth is not complete, but clearly it is not meant to be; from the very beginning we placed restrictions on the size of domains involved in the specification to be able to turn first-order formulas into propositional formulas. This is just another step in the same direction.

One interesting feature of our proposal, is that there are witnesses for the intermediate states of the counterexample trace. This is due to the fact that the translation we presented introduces fresh variables for each value update.

Chapter 4

DynJML: a relational object-oriented language

DynJML is a relational specification language originally created as an intermediate representation for the translation from JML [34] specifications into DynAlloy models. Its relational semantics made DynJML an appropriate target for other formalisms such as JFSL [91] and AAL [56].

Like Jimple [83], DynJML is an object-oriented language whose syntax is much more simpler than other OO languages such as Java or C#. As we will see, this allows us to implement a more compact and elegant translation to DynAlloy. As with Java and Jimple, there is a clear procedure for translating Java (annotated with JML) code into a DynJML equivalent.

We say that DynJML is a relational specification language since every expression is evaluated to a set of tuples. Even though it is not an extension, DynJML has the same type system, expressions and formulas that Alloy has.

Appendix A shows DynJML grammar. In the remaining of this section we will review DynJML grammar and semantics by means of a motivating example.

4.1 DynJML syntax and semantics

Signatures are declared in DynJML using Alloy notation. In Listing 4.1 it is shown how to declare a signature named *java_lang_Object*. We will treat every atom belonging to this signature as an object stored in the memory heap.

Listing 4.1. Declaring a signature

```
1 sig java_lang_Object {}
```

Suppose we want to write an abstract data type for a set of objects. This ADT will be implemented using a acyclic singly linked list. We may start extending the *java_lang_Object* signature for characterizing the linked list's node objects, as shown in Listing 4.2.

Listing 4.2. Extending a signature

```
1 sig Node extends java_lang_Object {
2   value : one java_lang_Object ,
3   next  : one Node+null
4 }
```

We define two different Alloy fields. Field *value* is intended to maintain a reference to the element stored in the set container. Field *next* is intended to store the following Node in the sequence (or value *null* in case no such Node exists). In high level programming languages such as Java or C#, *null* is a distinguished value. It indicates reference to no object. To allow a reference to no object, DynJML provides a predefined singleton signature named *null*. This signature cannot be extended.

Although the same semantics may be accomplished by declaring the field as *lone*, using a distinguished atom *null* helps achieving a more compact and elegant translation from Java or C# into DynJML.

Once the Node signature is declared, we may continue by defining the signature for *LinkedSet* objects (Listing 4.3). The sole field for this signature is intended to mark the beginning of the acyclic sequence of Node elements. *null* is included in the field image since the empty set will be represented as a *LinkedSet* object whose *head* field points to no Node object.

Listing 4.3. Declaring the LinkedSet signature

```
1 sig LinkedSet extends java_lang_Object {
2   head : one Node+null
3 }
```

Once signatures were introduced, we turn our attention to declaring programs. In Listing 4.4 a program for testing membership is shown.

Listing 4.4. A program for testing set membership

```
1
2 program LinkedSet::contains[thiz   : LinkedSet ,
3                             elem   : java_lang_Object+null ,
4                             return : boolean] {
```

```
5     Implementation
6     {
7       return := false ;
8       var current : one Node + null ;
9       current := thiz.head ;
10      while (return == false && current != null) {
11        if (current.value == elem) {
12          return := true ;
13        } else {
14          current := current.next ;
15        }
16      }
17    }
18  }
```

DynJML syntax allows common control-flow constructs such as conditionals and loops. It also allows declaring local variables (for instance, variable *current*). Fields and variables may be updated using an assignment statement. As shown in the listing, another predefined signature in DynJML is *boolean*. This abstract signature is extended with two singleton signatures: *true* and *false* that serve as boolean literals.

Due to DynJML procedural flavour, the convention for representing the implicit receptor object is to explicitly declare a formal parameter named *thiz*. Note that DynJML uses the name *thiz* instead of *this* since the latter is a reserved word in Alloy. Following this convention, a static program differs from a non-static program since it has no formal parameter *thiz*.

Specifying program behaviour

So far we have been able to declare signatures (which may be seen as object-oriented classes) and programs (which may also be seen as object-oriented methods).

One interesting feature of DynJML is that it allows the specification of the behaviour of a program. In Listing 4.5 program *contains* is augmented with Alloy formulas specifying its behaviour.

Listing 4.5. Declaring a signature

```
1
2  program LinkedSet::contains[thiz   : LinkedSet,
3                              elem   : java_lang_Object+null,
```

```
4                                   return: boolean] {
5   Specification
6   {
7    SpecCase #0 {
8     requires { some n: Node | { n in thiz.head.*next-null
9                                   and n.value==elem } }
10    modifies { NOTHING }
11    ensures { return'==true }
12   }
13   and
14   SpecCase #1 {
15    requires { no n: Node | { n in thiz.head.*next-null
16                              and n.value==elem } }
17    modifies { NOTHING }
18    ensures { return'==false }
19      }
20   }
21   Implementation {   ... }
22
23  }
```

The relation between input and output state for program *contains* is characterized by introducing *SpecCase* clauses. Every *SpecCase* clause may possibly define a particular input-output mapping. The *requires* clause states what the memory heap and actual arguments should conform at program invocation. The *modifies* clause states which memory locations may be changed by the program. Finally, the *ensures* clause captures how program state evolves when program execution finishes. In order to characterize the input and output state, and the relation among both states, we may use the full expressive power of Alloy formulas. For instance, in Listing 4.5 quantification and reflexive transitive closure, and set difference are used.

In the shown example, two different specification cases are written. The first one states that, if a node exists such that it is reachable from the LinkedSet's *head* navigating the *next* field, and its *value* field points to the *elem* parameter, the value for parameter *return* must be equal to *true* at program exit. Like DynAlloy, we refer to a field or variable value in the output state by adding an apostrophe (in the example: *return'*). Analogously to the first specification case, the second one states that if no such node exists, the value of variable *return* at program exit must be equal to *false*.

The reserved keyword *NOTHING* states that no field should be updated by this program. In the example, both specification cases declare no field updating will be performed.

Given a program specification, the program precondition is established by the conjunction of the *requires* clauses defined in each specification case. It is not mandatory for specification cases to characterize disjoint input states. Notice that, a program precondition characterizing only a subset of all possible input states is absolutely legal.

Abstractions

A useful mechanism for writing specifications is *abstraction*. Recall the *contains* specification. Let us assume that the LinkedSet implementation is modified introducing an array instead of the Node sequence for storing the references to the set elements. In such scenario, it is necessary to update the specification for program *contains* to reflect the new implementation.

The construct *represents* allows us to map the *concrete* implementation values to some *abstract* value for a field. The sole restriction to such mapping is that abstract fields may only be accessible from within behaviour specifications. Any change to the implementation will be limited to changing the *represents* clause.

Listing 4.6. Declaring a represents clause

```
1  sig LinkedSet extends java_lang_Object {
2    head    : one Node+null,
3    mySet   : set java_lang_Object
4  }
5
6  represents LinkedSet::mySet such_that {
7    all o: java_lang_Object+null | {
8      o in thiz.mySet
9      iff some n : Node | n in thiz.head.*next-null and n.value=o
10   }
11 }
```

Now, the specification for program *contains* may be rewritten referring to the *mySet* abstract field as shown in Listing 4.7.

Listing 4.7. Rewriting contains specification

```
1
2  program LinkedSet::contains[thiz   : LinkedSet,
```

```
3                                   elem   : java_lang_Object+null ,
4                                   return: boolean] {
5    Specification
6    {
7       SpecCase #0 {
8         requires { elem in thiz.mySet }
9         modifies {}
10        ensures  { return'==true }
11      }
12      and
13      SpecCase #1 {
14        requires { elem !in thiz.mySet }
15        modifies {}
16        ensures  { return'==false }
17      }
18
19   }
20   Implementation {  ... }
21
22 }
```

An informal semantics for the *represents* construct is that the abstract field receives any value such that the *represents* condition holds. This semantics may be referred as *relational* abstraction in contrast to *functional* abstraction.

Notice that if the abstract field is accessed from a *requires* clause, its value will depend on the input state. If it is accessed from within a *ensures* and adding an apostrophe, it will depend on the state at program exit.

Object Invariants

In object-oriented programming, an invariant is a property that should hold in all states visible to the client of that object. It must be true when control is not inside the object's methods. That is, an invariant must hold at the end of each constructor's execution, and at the beginning and end of all methods. Invariants are present in a wide range of languages like JML, Spec# and Eiffel to name a few.

DynJML allows the definition of signature invariants following the same semantics. Given the set of Node elements reachable from the LinkedSet's head, it is required that:

- No Node element is reachable from itself by navigating the *next* field. This means no cycles are allowed in the next field.
- No pair of distinct Node elements exists such that they refer to the same object.

This condition is expressible using the *object_invariant* construct:

Listing 4.8. Declaring an object invariant

```
1  object_invariant LinkedSet {
2    all n : Node | {
3      n in thiz.head.*next - null
4      implies n !in n.next.*next
5    } and
6    all n1,n2: Node | {
7     (n1 != n2 and
8       n1 in thiz.head.*next -null and
9       n2 in thiz.head.*next -null)
10    implies n1.value != n2.value
11   }
12 }
```

Modifying the system state

Obviously, as any object-oriented program language, DynJML programs may modify fields apart as well as program arguments. Listing 4.9 shows the specification and implementation for program *remove*. In order to remove any element, it is required to update fields.

Listing 4.9. Specification and implementation of program remove

```
1
2  program LinkedSet::remove[thiz    : LinkedSet,
3                            elem    : java_lang_Object+null,
4                            return  : boolean] {
5    Specification
6    {
7       SpecCase #0 {
8         requires { elem in thiz.mySet }
9         modifies { thiz.head, Node.next }
```

```
10          ensures  { thiz.mySet' == thiz.mySet - elem &&
11                     return' == true }
12        } and
13        SpecCase #1 {
14          requires { elem !in thiz.mySet }
15          modifies { }
16          ensures  { return'==false }
17        } and
18    }
19    Implementation {
20      var previous : one Node + null ;
21      var current  : one Node + null ;
22      current  := thiz.head ;
23      previous := null ;
24      while (current != null) {
25        if (current.value == elem) {
26          if (previous==null) {
27            thiz.head := current.next;
28          } else {
29            previous.next := current.next;
30          }
31        } else {
32          current := current.next ;
33        }
34      }
35    }
36
37  }
```

In the previous listing we may see how by using Alloy expressions the set of possible update locations is defined.

A program may also specify that it may modify any field location. For stating this behaviour the reserved keyword *EVERYTHING* is used.

Memory allocation and program invocation

New atoms may be allocated in the memory heap by invoking the *createObject* statement as seen in Listing 4.10. On the other hand, the *call* statement invokes the execution of other declared programs.

Listing 4.10. Memory allocation and program invocation

```
1  virtual program LinkedSet::add[thiz    : LinkedSet,
2                                 elem    : java_lang_Object+null,
3                                 return  : boolean] {
4    Specification { .... }
5    Implementation {
6      var ret_contains : one boolean ;
7      call LinkedSet::contains[thiz, elem, ret_contains] ;
8      if (ret_contains==false) {
9        var new_node : one Node + null ;
10       createObject<Node>[new_node] ;
11       new_node.value := elem ;
12       new_node.next  := thiz.head ;
13       thiz.head := new_node ;
14       return := true ;
15     } else {
16       return := false ;
17     }
18   }
19
20 }
```

Notice that we have declared the *add* program using the modifier *virtual*. We will discuss the semantics of this keyword later.

Program inheritance

DynJML adds to Alloy's polymorphism the inheritance of programs when a signature is extended. For instance, we may define a new signature *SizeLinkedSet* extending signature *LinkedSet*. Alloy field *size* is intended to store the number of elements contained in the set.

Listing 4.11. Extending LinkedSet signature

```
1 sig SizeLinkedSet extends LinkedSet {
2   size: one Int
3 }
```

SizeLinkedList inherits all preconditions, postconditions, abstraction relations and invariants from LinkedSet. Nevertheless, the invariant for SizeLinkedList

must be augmented to add a constrain on the values for field size. The intended value for this field is the number of elements store in the set.

Listing 4.12. Augmenting an invariant

```
1  object_invariant SizeLinkedSet {
2    thiz.size = #(thiz.head.*next - null)
3  }
```

In object-oriented programming, a method is overridden if the subclass provides a new specific implementation to a superclass method. The same concept may be found in DynJML, programs may be overridden by a an extending signature as shown in Listing 4.13.

Listing 4.13. Overriding a program

```
1  program SizeLinkedSet::add[thiz   : SizeLinkedSet,
2                             elem   : java_lang_Object+null,
3                             return : boolean] {
4  Specification {
5    SpecCase#0 {
6      modifies { thiz.size }
7    }
8  }
9  Implementation {
10   var ret_val : one boolean ;
11   super_call LinkedSet::add[thiz,elem,ret_val] ;
12   if (ret_val==true) {
13     thiz.size := thiz.size + 1;
14   } else {
15     skip;
16   }
17   return := ret_val ;
18 }}
```

The *super_call* statement invokes the program at the extended signature. Notice that, while the implementation may be completely removed and replaced, the specification (as the invariant) only may be augmented. In other words, the description of the program behaviour may only grow in detail, but it is not possible to contradict its parent's specification.

In object-oriented programming, a virtual function or virtual method is a function or method whose behaviour can be overridden within an inheriting class by

a function with the same formal parameters. In DynJML, the *virtual* modifier allows those signatures extending the parent signature to override the program.

Program overloading

In order to alleviate the translation burden to DynAlloy from the high-level programming languages, another feature supported by DynJML is *overloading*. Method overloading allows the creation of several methods with the same name which differ from each other in terms of the typing of the formal parameters. In Listing 4.14 we overload program *addAll* by defining two versions, the first one deals with arguments of type SizeLinkedList while the second one receives arguments of type LinkedSet.

Listing 4.14. Overloading a program

```
1 program SizeLinkedSet::addAll[thiz : SizeLinkedSet,
2                               aSet : SizeLinkedSet+null]
3 {...}
4
5 program SizeLinkedSet::addAll[thiz : SizeLinkedSet,
6                               aSet : LinkedSet+null]
7 {...}
```

Abstract programs

A program declared as *abstract* has a specificatin but no implementation. An abstract DynJML program serves as the target for representing Java and C# abstract methods.

Listing 4.15. Abstract program declaration

```
1 abstract program AbstractSet::isEmpty[thiz : AbstractSet,
2                                       return : boolean]{
3   Specification {
4     SpecCase #0 {
5       ensures { return'==true iff some thiz.mySet }
6     }
7   }
8 }
```

Program invocation in specifications

A very useful feature for writing readable specifications is the capacity of *invoking* a program as term in the logical formula. DynJML follows the semantics presented in Cok [16]. for dealing with program invocation in specifications. As the reader may imagine, since specifications may not alter the memory state, the called program may be side-effect free, which is known as *pure* ([7]). An example may be found in Listing 4.16.

Listing 4.16. Procedure calls in specifications

```
1  program LinkedSet::contains[thiz : LinkedSet,
2                              return : boolean]
3  {...}
4
5  program LinkedSet::isEmpty[thiz : LinkedSet,
6                             return : boolean
7    Specification {
8      SpecCase #0 {
9        modifies { NOTHING }
10       ensures {
11         return' == true
12         iff some n : java_lang_Object+null |
13             spec_call LinkedSet::contains[thiz,true]
14       }
15     }
16   }
17 }
```

A program may be invoked in a specification if and only if

- the program has no side-effects
- the only argument it modifies is named *return*

Only under those circumstances a program may be invoked within a DynJML specification.

Assertions

The *assert* statement allows the specification writer to include a formula that *must* hold in a given location in the program implementation. Since these condi-

tions are intended to hold if the program control-flow executes that location, we may see them as part of the specification embedded within the implementation.

Assertion statements are commonplace for several specification languages such as JML and Spec#. Assertions allow the writer to predicate on intermediate program states beyond the pre-state and post-state. What is more, assertions may reference to local variable which are not accessible from the specification.

Listing 4.17. The assertion statement

```
1 var ret_val : boolean ;
2 var list: LinkedSet+null ;
3 createObject<LinkedSet>[list] ;
4 call LinkedSet::add[list,elem,ret_val] ;
5 assert ret_val==true ;
6 call LinkedSet::remove[list,elem,ret_val] ;
7 assert ret_val==false
```

Notice that, since assertions are intended to alter the structured control-flow, a DynJML program including assertions statements may not be as easily transformed into a DynAlloy as a DynJML program with no assertions.

The approach we choose for dealing with assertions was to perform a pre-translation phase before actually passing the DynJML program to the translator. This phase transforms the DynJML program P into an equivalent program P' where assertions statements are replaced and structured control-flow is restored. We will show the details in Section 4.2.

Loop invariants

Like Spec#, JML and Eiffel, loop invariants may be written in DynJML. Informally, a loop invariant is a condition that should hold on entry into a loop and that must be preserved on every iteration of the loop. This means that on exit from the loop both the loop invariant and the loop termination condition can be guaranteed.

The following Listing shows an example of a loop annotated with an invariant condition:

Listing 4.18. Loop invariants

```
1 program LinkedSet::count[thiz: LinkedSet, return: Int] {
2   Specification {
3     SpecCase #0 {
4       requires { true }
```

```
5         modifies { NOTHING }
6         ensures  { return'==#(thiz.head.*next - null)
7       }
8     }
9     Implementation {
10      return := 0 ;
11      var curr : Node + null ;
12      curr := thiz.head ;
13      while (curr!=null)
14      loop_invariant #(curr.*next - null) + return
15                     == #(thiz.head.*next - null)
16      {
17        return := return + 1 ;
18        curr := curr.next ;
19      }
20    }
21  }
```

The Assume and Havoc Statements

Many intermediate representations for program analysis (such as the one used by ESC/Java2, BoogiePL used by Spec#, and FIR used by JForge) offer support for assume and havoc statements.

While the verification of an *assert* statement fails if the condition is not met. The *assume*, on the contrary, coerces a particular condition to be true. Namely, only those models where the condition holds are considered during verification. The introduction of assumptions in the specification languages obeyed to two different goals:

- Allow the addition of redundant conditions in order to help static analysis tools.
- Allow the end-user to specify a particular set of executions for analysis.

A *havoc* statement specify an expression whose value may change non-deterministically. By combining a havoc and an assume statement, the value of an expression may change non-deterministically to satisfy a given condition.

Listing 4.19. The havoc/assume statements

```
1   havoc  x ;
```

```
2    assume x>0 ;
```

If the expression being *havocing* corresponds to a field access, the intended semantics for this statement is that of *updating* the field, but not the receiver.

Listing 4.20. Havocing a reference

```
1    havoc  thiz.size ;
2    assume thiz.size>0 ;
```

4.2 Analyzing DynJML specifications

In this section we will see how DynJML is analyzed by translating DynJML specifications into DynAlloy models. It will be made clear in this section that once DynAlloy is available, translating DynJML becomes immediate.

In order to handle aliased objects appropriately we adopt relational view of the heap from JAlloy [52]. In this setting, types are viewed as sets, fields as binary functional relations that maps elements of their class to elements of their target type; and local variables as singleton sets. Under the relational view of the heap, field dereference becomes relational join and field update becomes relational override.

As already stated, one key goal while designing DynJML was keeping the language as close as possible to DynAlloy syntax. Due to this, we will see that DynJML specifications are translated smoothly into DynAlloy partial correctness assertions.

Adding signatures to DynAlloy

As we have said, the *null* value is represented as an Alloy singleton signature. Likewise, *boolean* literals are defined as singleton signatures extending an abstract *boolean* signature.

```
one sig null {}
abstract sig boolean {}
one sig true,false extends boolean {}
```

For every user-defined signature S, a signature without fields is defined, since fields will be explicitly passed as arguments.

Recalling the LinkedSet example, the following signatures are defined:

```
sig java_lang_Object {}
sig Node extends java_lang_Object {}
sig LinkedSet extends java_lang_Object {}
sig SizeLinkedSet extends LinkedSet {}
```

The following binary relations for modelling fields will be passed as arguments when required :

```
next : Node -> one (Node+null)
value: Node -> one java_lang_Object
head : LinkedSet -> one (Node+null)
mySet: LinkedSet -> java_lang_Object
size : SizeLinkedSet -> Int
```

Modeling actions

Binary relations can be modified by DynAlloy actions only. We will in general distinguish between simple data that will be handled as values, and structured objects.

Action *update_reference* is introduced to modify a given object's field:

```
action field_update[field: univ->univ,
                    left : univ,
                    right: univ] {
  pre  { true }
  post { field' = field ++ (left->right) }
}
```

In order to translate assignment of an expression to a variable, we introduce action *variable_update* as follows:

```
action variable_update[left: univ, right: univ] {
  pre  { true }
  post { l' = r }
}
```

We introduce now in DynAlloy an action that allocates a fresh atom. We denote by *alloc_objects* the unary relation (set) that contains the set of objects alive at a given point in time. This set can be modified by the effect of an action. In order to handle creation of an object of concrete type C in DynAlloy, we introduce an action called *alloc* specified as follows:

```
action alloc[alloc_objects: set univ,
             fresh : univ,
             typeOf: set univ] {
  pre  { true }
  post { fresh' in typeOf &&
         fresh' !in alloc_objects &&
         alloc_objects' = alloc_objects + fresh' }
}
```

Notice that as parameter *typeOf* should receive the target set for the object concrete type, we are able to use the same action for allocating any concrete type.

Some variables might need to change non-deterministically due to the semantics of statements such as havoc or an abstract field. In order to represent non-deterministic change, we introduce a DynAlloy action to (possibly) *erase* the value of a variable:

```
action havoc_variable[a: univ] {
  pre  { true }
  post { a' = a' }
}
```

This (apparently) harmless action performs a subtle state change. By asserting something about argument *a* at the post-condition, it introduces a new value for variable a. Notice that, since the post-condition is a tautology, no constraint is imposed on this new value (besides its type).

As variables and also references may be *havoced*, a second action is defined to deal with *erasing* a field value for a given object:

```
action havoc_reference[f: univ -> univ, u: univ] {
  pre { true }
  post { u<:f = u<:f' && some u.f'  }
}
```

Translating a program implementation

Given a DynJML program implementation with no assertion statements and no loop invariants, translation function T creates the correspondent DynAlloy program *P*. The translation for non recursive statements is defined below. For clarification purposes we separate the assignment statement for updating a variable from the statement that updates a field.

T(v := expr1)	→	update_variable[v,expr1]
T(expr1.f := expr2)	→	update_reference[f, expr1, expr2]
T(createObject<T>[v])	→	alloc[alloc_objects,v,T]
T(skip)	→	(true)?
T(var l: *T*)	→	declare l:*T* as new local variable in DynAlloy program *P*
T(assume B)	→	(B)?
T(havoc v)	→	havoc_variable[v] (v is a variable)
T(havoc expr1.f)	→	havoc_reference[f, expr1]

For more complex program constructs, the translation is defined as follows:

T(while pred { stmt })	→	(pred?;T(stmt))*;(!pred)?
T(if pred { stmt1 } else { stmt2 })	→	(pred?;T(stmt1)) + (!pred?;T(stmt2))
T(stmt1 ; stmt2)	→	T(stmt1) ; T(stmt2)

As we can see, since DynJML expressions are in fact Alloy expressions, the translation to DynAlloy is compact and elegant. The DynAlloy program declaration is completed by:

- Copying each formal parameter declared in the DynJML program.
- For each binary relation *F* in the DynAlloy program, a formal parameter *F* with the corresponding type is declared.

Recalling program *contains* from Listing 4.4, the DynAlloy program becomes:

```
program contains_impl[thiz : LinkedSet,
                      elem : java_lang_Object+null,
                      return: boolean ,
                      head : LinkedSet -> one (Node+null),
                      next : Node -> one (Node+null),
                      value: Node -> java_lang_Object]
local [current: Node+null]
{
  update_variable[return, false] ;
  update_variable[current, thiz.head] ;
  (
    (return==false && current != null)? ;
    ((current.value==elem)?;
```

```
      update_variable[return,true])
    +
    (!(current.value==elem)?;
      update_variable[current,current.next])
  )* ;
  !(return==false && current != null)?
}
```

Partial Correctness Assertions

The basic idea for analyzing DynJML is to transform the specification and the program implementation into a DynAlloy partial correctness assertion. If the assertion is invalid, a violation to the specification occurs. On the other hand, if the assertion is valid, no violation to the specification exists *within* the scope of analysis. We will discuss what we understand by *scope* of analysis in the next section.

Every specification case S_i consists of:

- a requires formula $Req(\overline{x})$,
- a set of locations that may be modified $\{expr_1.f_1, \ldots, expr_n.f_n\}$, and
- an ensures formula $Ens(\overline{x}, \overline{x}')$.

For a every specification case S_i a logical formula α_{S_i} is defined as follows:

$$
\begin{aligned}
Req(\overline{x}) \quad \Rightarrow (& \\
& Ens(\overline{x}, \overline{x}') \wedge \\
& (\forall o)(o \in Dom(f_1) \wedge o.f_1 \neq o.f_1' \Rightarrow o \in expr_1) \wedge \\
& \ldots \\
& (\forall o)(o \in Dom(f_n) \wedge o.f_n \neq o.f_n' \Rightarrow o \in expr_n) \\
) &
\end{aligned}
$$

This formula states that, if the requires condition holds at the input state $\overline{x}$ (represented as a vector of relations), then :

- the ensures condition should hold at the output state $\overline{x}'$

- For each location $expr_i.f_i$ in the modifies clause, if the field f_i was modified, then the modified location belongs to the set that arises from evaluating $expr_i$

Given $\{g_1, \ldots g_q\}$ the set of fields such that g_i is not present in any modifies clause, we extend the previous formula stating that these fields cannot be modified:

$$
\begin{aligned}
Req(\overline{x}) \quad \Rightarrow (& \\
& Ens(\overline{x}, \overline{x}') \wedge \\
& (\forall o)(o \in Dom(f_1) \wedge o.f_1 \neq o.f_1' \Rightarrow o \in expr_1) \wedge \\
& \ldots \\
& (\forall o)(o \in Dom(f_n) \wedge o.f_n \neq o.f_n' \Rightarrow o \in expr_n) \wedge \\
& (\forall o)(o \in Dom(g_1) \wedge o.g_1 = o.g_1') \wedge \\
& \ldots \\
& (\forall o)(o \in Dom(g_q) \wedge o.g_q = o.g_q') \\
)
\end{aligned}
$$

Given the specification cases $\{S_1, \ldots S_m\}$, the corresponding Alloy formulas $\{\alpha_{S_i}(\overline{x}, \overline{x}'), \ldots, \alpha_{S_m}(\overline{x}, \overline{x}')\}$ are defined. We cojoin these formulas with the DynAlloy program $P[\overline{x}]$ obtained from translating the program implementation. The resulting DynAlloy partial correctness assertion is defined as follows:

$$
\begin{aligned}
& \{true\} \\
& P[\overline{x}] \\
& \{\alpha_{S_1}(\overline{x}, \overline{x}') \;\&\&\; \ldots \;\&\&\; \alpha_{S_m}(\overline{x}, \overline{x}')\}
\end{aligned}
$$

Notice that, since our intention is to denote the set of objects alive with variable *alloc_objects*, it is required to state that this variable contains all atoms reacheable in the pre-state. Given a program declaration $M[p_1, \ldots, p_l]$ we will say that an atom a is *allocated* if:

- a is equal to p_i
- a is reacheable from some allocated atom b using some field f.

Let $\{f_1, \ldots f_j\}$ be the set of binary relations representing fields, let $\{p_1, \ldots p_l\}$ be the collection of formal parameters in the DynJML program, the following expression Σ characterizes the set of allocated atoms. Observe that *null* is explicitly restrained from this set.

$$(p_1 + \cdots + p_l).*(f_1 + \cdots + f_j) - \text{null}$$

We may refer to y as an abbreviation for *alloc_objects*. Now we can state that we are only interested in those models where y at the precondition is equal to the set of all allocated objects:

$$\begin{array}{l} \{ \quad y = \Sigma \quad \} \\ P[\overline{x}, y] \\ \{ \quad \alpha_{S_1}(\overline{x}, \overline{x}') \ \wedge \cdots \wedge \ \alpha_{S_m}(\overline{x}, \overline{x}') \quad \} \end{array}$$

As already stated, invariants are conditions that must be preserved by the program under analysis. The sole free variable in an invariant condition (besides field references) is the *thiz* variable. Observe that invariants may be asserted only over a set of allocated objects. For each invariant $Inv_T(thiz, \overline{x})$ for signature T, we define a formula $\beta_T(\text{alloc_objects}, \overline{x})$ stating that the invariant holds for all allocated objects of type T:

$$(\forall o)(o \in T \cap y \Rightarrow Inv_T(o, \overline{x}))$$

Since objects may be deallocated during the execution of the program, we need to update the value of variable y at the post-state. This is done via the *havoc* action. Now by combining this action with the test predicate, we can constraint variable y to which values it may store at the post-state.

For each signature T in $\{T_1, \ldots, T_k\}$, the $\beta_T(\overline{x}, y)$ condition is assumed in the pre-state and asserted in the post-state as follows:

$$\begin{array}{l} \{ \quad y = \Sigma \quad \wedge \\ \quad \beta_{T_1}(\overline{x}, y) \ \wedge \ \ldots \ \wedge \ \beta_{T_k}(\overline{x}, y) \quad \} \\ P[\overline{x}, y] \ ; \\ \mathit{havoc_variable}[y] \ ; [y = \Sigma]?; \\ \{ \quad \alpha_{S_1}(\overline{x}, \overline{x}') \ \wedge \cdots \wedge \ \alpha_{S_m}(\overline{x}, \overline{x}') \ \wedge \\ \quad \beta_{T_1}(\overline{x}', y') \ \wedge \ \ldots \ \wedge \ \beta_{T_k}(\overline{x}', y') \quad \} \end{array}$$

Observe that, while the invariant is assumed on all objects allocated at the pre-state, it is asserted on all objects allocated at the *post-state*. This is achieved by referring to y at the post-state (namely, y').

Finally, as we have seen, abstractions are supported by DynJML. An abstraction is introduced in DynJML using the *represents* clauses. Each *represents* includes:

- a field a storing abstract values that is constrained, and
- an abstraction predicate γ that links concrete values $\overline{x}$ to the field a.

Recall that the sole restriction to abstract fields was that they were only accessible from specifications. This means that no abstract field is referenced in the program implementation, only at the pre-state and at the post-state.

For each field storing abstract values in $\{a_1, \ldots a_t\}$, we have $\gamma_{a_i}(\overline{x})$ the abstraction predicate that constrains the abstract values. We follow the same mechanism used for the allocated objects: we assume the predicate at the pre-state, and we *havoc* the field value after the program execution.

Finally, we conclude the definition of the partial correctness assertion to analyze a DynJML program as follows:

$$
\begin{array}{l}
\{ \quad y = \Sigma \quad \wedge \\
\quad \beta_{T_1}(\overline{x}, y) \ \wedge \ \ldots \ \wedge \ \beta_{T_k}(\overline{x}, y) \wedge \\
\quad \gamma_{a_1}(\overline{x}) \wedge \ \ldots \ \wedge \gamma_{a_t}(\overline{x}) \quad \} \\
P[\overline{x}, y] \ ; \\
\mathit{havoc_variable}[y] \ ; [y = \Sigma]? \ ; \\
\mathit{havoc_reference}[a_1, \mathit{thiz}] \ ; [\mathit{thiz}.a_1 = \gamma_{a_1}(\overline{x})]? \ ; \\
\ldots \\
\mathit{havoc_reference}[a_t, \mathit{thiz}] \ ; [\mathit{thiz}.a_t = \gamma_{a_t}(\overline{x})]? \ ; \\
\{ \quad \alpha_{S_1}(\overline{x}, \overline{x}') \ \wedge \cdots \wedge \ \alpha_{S_m}(\overline{x}, \overline{x}') \wedge \\
\quad \beta_{T_1}(\overline{x}', y') \ \wedge \ \ldots \ \wedge \ \beta_{T_k}(\overline{x}', y') \quad \}
\end{array}
$$

Notice that, if fields $\{a_1, \ldots a_t\}$ are navigated at the pre-state, the value they will store is the intended since conditions $\gamma_{a_1}(\overline{x}) \wedge \ \ldots \ \wedge \gamma_{a_t}(\overline{x})$ are assumed in the precondition. Similarly, if fields $\{a_1, \ldots a_t\}$ are accessible at the post-state, erasing the field and assuming its value holds the abstraction predicate, leads to the intended abstract value at the program exit.

Procedure calls

Program invokation is transformed into DynAlloy procedure calls. As we have previously seen, DynAlloy supports procedures calls using the *call* statement. Nevertheless, since DynAlloy does not support neither overloading nor

overriding, the translation from DynJML to DynAlloy must bridge this semantic gap.

As most compilers, translating DynJML involves statically resolve overloading and non-virtual procedure calls. In order to do so, the translation to DynAlloy statically renames every program with a unique identifier. Given a program name *m* declared within signature *S*, it is renamed by

- adding the signature name S as a prefix, and
- adding an integer number i as a suffix if the program is overloaded. The value for i corresponds to the syntactic order in which the program is declared in the DynJML source.

It is easy to see that if two non-virtual programs have the same identifier, this renaming would avoid any ambiguity. As an example, the following DynAlloy program declarations result from translating DynJML programs listed in Listing 4.14 :

```
program SizeLinkedSet_addAll_0[...]
{...}

program SizeLinkedSet_addAll_1[...]
{...}
```

As we have stated, resolving a non-virtual and/or overloaded procedure call statically is commonplace for today's compilers. On the other hand, compilers resolve virtual procedure calls by creating and maintaining virtual function tables [3]. These function tables are used at runtime for selecting the actual program based on the types of parameters. This mechanism is known as dynamic dispatch [4].

The mechanism for representing dynamic dispatch in DynAlloy is borrowed from VAlloy [64]. In VAlloy, a formula simulating the dynamic dispatch is built. We will apply the same technique but using a DynAlloy program instead of an Alloy formula. As we will see, this program will state that the actual program may be invoked if and only if the receiver parameter (namely, *thiz*) strictly belongs to the signature.

First the signature hierarchy is computed. Given a signature hierarchy H, a signature S' belongs to $children(S)$ if and only if S is an ancestor of S'. Using this definition, we can build the following relational term denoting only those atoms that strictly belong to S and do not belong to any descendent of S:

$$S - (\bigcup_{S' \in children(S)} S')$$

Generally, given a virtual program m in signature S_0 such that $children(S_0) = \{S_1, \ldots, S_n\}$, the program forcing the runtime dispatching is defined as follows:

```
program virtual_m[thiz: S0,...] {
  [thiz in (S0-(/*union of descendants of S0*/)]?;
     call S0_m[thiz,...]
  +
  [thiz in (S1-(/*union of descendants of S1*/)]?;
     call S1_m[thiz,...]
  +
  ...
  +
  [thiz in (Sn-(/*union of descendants of Sn*/)]?;
     call Sn_m[thiz,...]
}
```

Let us show that all test actions are exclusive by means of a proof by contradiction. Let us consider that atom a belongs simultaneously to $S_i - children(S_i)$ and $S_j - children(S_j)$. If S_i and S_j are descendants of S there are only three possible cases to consider:

- S_i is an ancestor of S_j
- S_j is an ancestor of S_i
- S_i and S_j do not extend each other, but both are descendants of a common ancestor S'.

Let us consider the case where S_i is an ancestor of S_j. Then, S_j belongs to $children(S_i)$. If a belongs to $S_i - children(S_i)$, then it holds that $a \in S_i$ and $a \notin children(S_i)$. Therefore, $a \notin S_j$, and consequently $a \notin S_j - children(S_j)$, which is a contradiction. The proof for S_j is an ancestor of S_i follows analogously.

For the case where S_i and S_j do not extend each other, but $\{S_i, S_j\} \subset S'$, it is easy to see that S_i, S_j are not disjoint due to $a \in S_i \cap S_j$.

As an example, as the computed signature hierarchy is {SizeLinkedSet $\subset$ LinkedSet}, the translation outputs the following program for virtual program in Listing 4.10.

```
program virtual_LinkedSet_add[thiz: LinkedSet, ...] {
  [thiz in (LinkedSet - SizeLinkedSet)]?;
     call LinkedSet_add[thiz,...]
  +
  [thiz in SizeLinkedSet]?;
     call SizeLinkedSet_add[thiz,...]
}
```

Transforming Assertion statements

Assertions statements alter the structured control-flow which complicates the translation from DynJML programs into DynAlloy. To avoid this, a DynJML program with assertions is transformed into another equivalent DynJML program free of assertion statement. The transformation works as follows:

1. a fresh unused boolean variable (namely, *assertion_failure*) is declared as an input parameter in every program.
2. Any statement *assertion* α is replaced the statement: *if (assertion_failure==false* $\wedge\neg\alpha$ *) then assertion_failure:=true endif.*
3. Any non-recursive statement *P* is *guarded* by replacing it with statement : *if assertion_failure==false then P endif.*
4. The program precondition is augmented with condition *assertion_failure==false*, stating that no assertion is violated at the pre-state.
5. Condition *assertion_failure'==false* is added to the program postcondition.

As an example, recall Listing 4.17. Parameters for both programs *remove* and *add* are modified by including the *assertion_failure* boolean. The transformed program produced for that input follows:

Listing 4.21. An assertion-free program

```
1 var ret_val : boolean ;
2 var list: LinkedSet+null ;
3 if assertion_failure==false {
4   createObject<LinkedSet>[list] ;
5 }
6 if assertion_failure==false {
```

```
7     call LinkedSet::add[list,elem,ret_val,assertion_failure];
8   }
9   if assertion_failure==false and ret_val!=true {
10    assertion_failure:=true;
11  }
12  if assertion_failure==false {
13    call LinkedSet::remove[list,elem,ret_val,assertion_failure];
14  }
15  if assertion_failure==false and ret_val!=false {
16    assertion_failure:=true;
17  }
```

As the reader may think, the *assertion_failure* variable stores if any assertion was not satisfy. In that case, no further statements are evaluated. The value for the boolean variable in the pre-state is *false* due to the inclusion of this condition in the precondition. On the other hand, if an assertion condition did not hold at a given location, the *assertion_failure'*==*false* condition at the post-state triggers a violation of the program specification embedded within the implementation.

Transforming Loop invariants

Another feature that was explicitly excluded from the translation presented previously were loop invariants. Following the same approach applied to assertion statements, loop invariants are transformed into an equivalent DynJML program before translating it to DynAlloy.

Given a generic loop annotated with an invariant as the one presented below:

Listing 4.22. A generic loop invariant

```
1  while B
2  loop_invariant I {
3    S
4  }
```

We transform the loop above into the following sequence of statements:

Listing 4.23. Transforming a generic loop invariant

```
1  assert I;
2  havoc T;
3  assume I;
4  if B {
```

```
5    S ;
6    assert I ;
7    assume false ;
8  }
```

where S is the loop body, T are the locations updated by S. The predicate I serves as a loop invariant. Similarly to the translation applied in Spec#, the transformation causes the loop body to be verified in all possible states that satisfy the loop invariant. The assume false; command indicates that a code path that does not exit the loop can be considered to reach terminal success at the end of the loop body, provided that the loop invariant has been re-established.

The algorithm for statically computing the set of locations in T works as follows: First, all locations being updated are recollected. Secondly, if a location is a *expr.f* expression, and the receiver expression contains a variable or field reference whose value is being *havoced*, then *expr.f* location is replaced by the *f* location. This means that all the values stored in field *f* may be changed by executing the loop body. Although this supposes a gross overapproximation of the actual set of updated locations, the same approach is taken by other tools such as the Spec# compiler.

Recall Listing 4.18, the set of locations T computed by the algorithm is {*curr*, *return*}. On the other hand, given the while statement shown in Listing 4.24 the resulting T equals to {*curr*, *thiz.size*}:

Listing 4.24. Computing the set of updated locations

```
1  while (curr!=null)
2  loop_invariant #(curr.*next - null) + thiz.size
3                      == #(thiz.head.*next - null)
4  {
5    thiz.size := thiz.size + 1 ;
6    curr := curr.next ;
7  }
```

Due to DynAlloy's type system, a new DynAlloy action must be defined in order to havoc all field references.

```
action havoc_field[f: univ -> univ] {
  pre { true }
  post { f'.univ = f.univ  }
}
```

With this new action at hand, the translation for DynJML statements is extended with:

$$\text{T(havoc f)} \quad \rightarrow \quad \text{havoc_field[f] (f is a field)}$$

Modular SAT-based Analysis

In the presence of specifications for invoked programs, the analysis uses this specification as a *summary* of the invoked program. In other words, the invoked program implementation is assumed to obey its specification. This is known as *modular* analysis.

The specification may be also known as the *contract* of the program, because it states what are the program requirements at invocation and what clients may assume from its execution. In the context of SAT-based analysis this is known as modular SAT-based analysis.

Given a program specification for an invoked program P, DynJML transforms that specification into a implementation by replacing the original implementation for P with the following sequence of DynJML statements:

Listing 4.25. Computing the set of updated locations

```
1  Implementation {
2    assert R1 or ... or Rn ;
3
4    havoc M1 ;
5    ...
6    havoc Mn ;
7
8    assume R1 implies (E1 and FrameCondition_1) ;
9    ...
10   assume Rn implies (En and FrameCondition_n) ;
11 }
```

where $\{R_1 \ldots R_n\}$ are all *requires* clauses, $\{M1, \ldots, M_n\}$ is the conjunction of all *modifies* clauses, and $\{E_1, \ldots, E_n\}$ is the set of all *ensures* clauses. The $FrameCondition_i$ refers to the logical formula that states that only those locations specify in the modifies clause of the i-th specification case may change its value.

Since the specification are possibly partial, the modular analysis is not as precise as the whole program analysis. However, using the specifications instead of the actual implementation leads to a simpler DynAlloy model which could be more easily analyzed.

Chapter 5

TACO: from JML to SAT

TACO (Translation of Annotated COde) is the prototype tool we have built to implement the techniques presented in this dissertation. In this section we present an outline of TACO. This tool translates JML [34] annotated Java code to a SAT problem. This translation is in intention not much different from translations previously presented by other authors [28]. A schematic description of TACO's architecture that shows the different stages in the translation process is provided in Fig. 5.1.

TACO uses Alloy [49] as an intermediate language. This is an appropriate decision because Alloy is relatively close to JML, and the Alloy Analyzer [49] provides a simple interface to several SAT-solvers. Also, Java code can be translated to DynAlloy programs [39]. DynAlloy [35] is an extension of Alloy that allows us to specify actions that modify the state much the same as Java statements do. We will describe DynAlloy extensively in Chapter 3. DynAlloy's action behavior is specified by pre and post conditions given as Alloy formulas. From these atomic actions we build complex DynAlloy programs modeling sequential Java code.

As shown in Fig. 5.1 the analysis receives as input an annotated method, a scope bounding the sizes of object domains, and a bound LU for the number of loop iterations. JML annotations allow us to define a method contract (using constructs such as requires, ensures, assignable, signals, etc.), and invariants (both static and non-static). A contract may include normal behavior (how does the system behave when no exception is thrown) and exceptional behavior (what is the expected behavior when an exception is thrown). The scope constrains the size of data domains during analysis.

TACO architecture may be described as a pipeline following the translations described below:

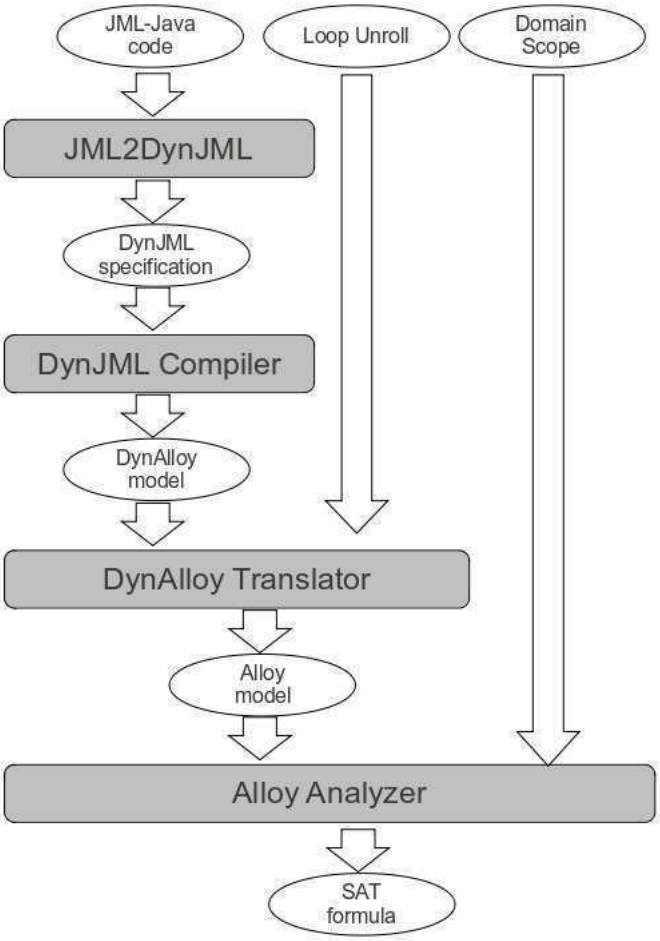


Figure 5.1. Translating annotated code to SAT

1. TACO begins by translating JML-annotated Java code into DynJML specifications using layer JML2DynJML. The DynJML language is a relational object-oriented language that bridges the semantic gap between an object-oriented programming language such as Java and the relational specification language DynAlloy.

2. The DynJML specification is then translated by the DynJML compiler into a single DynAlloy model using a rather straightforward translation [39]. This model includes a partial correctness assertion that states that every terminating execution of the code starting in a state satisfying the precondition and the class invariant leads to a final state that satisfies the postcondition and preserves the invariant.

3. The DynAlloy translator performs a semantically preserving translation from a DynAlloy model to an Alloy model. In order to handle loops we constrain the number of iterations by performing a user-provided number of loop unrolls. Therefore, the (static) analysis will only find bugs that could occur performing up to LU iterations at runtime.

4. Finally, the Alloy model is translated into a SAT formula. In order to build a finite propositional formula a bound is provided for each domain. This represents a restriction on the precision of the analysis. If an analysis does not find a bug, it means no bug exists within the provided scope for data domains. Bugs could be found repeating the analysis using larger scopes. Therefore, only a portion of the program domain is actually analyzed. Notice that an interaction occurs between the scope and LU. This is a natural situation under these constraints, and similar interactions occur in other tools such as Miniatur [31] and JForge [27].

5.1 Java Modeling Language (JML)

In previous chapters we introduce DynAlloy, an extension to Alloy with procedural action, and DynJML: an object-oriented specification language that is analyzable using a translation to DynAlloy specifications. Now we intend to introduce a tool that translates JML annotated code into DynJML.

As described in Leavens et al. [61], the Java Modeling Language (JML) is a behavioural interface specification language. JML can be used to specify the behaviour of Java programs. It combines the design by contract approach of Eiffel [66] and the model-based specification approach of the Larch [89] fam-

ily of interface specification languages, with some elements of the refinement calculus [5].

Since JML aims at bridging the gap between writing a program and writing its specification, Java expressions can be used as predicates in JML. However, as predicates are required to be side-effect free, only side-effect free Java expressions are valid.

We will walk through the JML syntax and semantics by means of linked list implementation annotated with JML that may be downloaded from the JML-Forge website [54]:

Listing 5.1. Annotating fields in JML

```
1  class LinkList {
2
3      static class Node {
4          /*@ nullable @*/ Node next;
5          /*@ nullable @*/ Node prev;
6          /*@ non_null @*/ Object elem;
7      }
8
9      /*@ nullable @*/ Node head;
10     /*@ nullable @*/ Node tail;
11     int size;
12 }
```

Notice first that text following `//` in a line, or text enclosed between `/*` and `*/`, is considered in Java as a comment. The JML parser, instead, considers a line that begins with `//@`, or text enclosed between `/*@` and `@*/` as a JML annotation. As JML annotations are introduced as Java comments this allows parsing any JML annotated Java source file with any parser complying the Java language specification.

The `nullable` modifier states that a given field may accept *null* as a valid value. However, the `non_null` modifier constrains all field values to non null. Contrary to programmer's intuition, by default fields are annotated as `non_null`.

Listing 5.2. A JML object invariant

```
1      /*@ invariant
2        @   (head == null && tail == null && size == 0)
3        @   ||
4        @   (head.prev == null && tail.next == null &&
```

```
5       @     \reach(head, Node, next).int_size() == size &&
6       @     \reach(head, Node, next).has(tail) &&
7       @     (\forall Node v; \reach(head, Node, next).has(v) ;
8       @                 v.next != null ==> v.next.prev == v));
9       @*/
```

The *invariant* clause allows the introduction of object invariants in the same way DynJML does. The intended semantics for invariants is explained in terms of visible states [62]. A state is a visible state for an object o if it is the state that occurs at one of these moments in a program's execution:

- at end of a constructor invocation that is initializing o,
- at the beginning or end of a method invocation with o as the receiver,
- when no constructor, method invocation with o as receiver is in progress.

Predicates are boolean valued expressions. To the boolean operators of negation (`!`), disjunction (`||`) and conjunction (`&&`) provided by Java, JML incorporates more operators such as logical implication (`==>`), logical equivalence (`<==>`), and universal (`\forall`) and existential (`\exists`) quantification. A formula of the form

$$(\texttt{\textbackslash forall}\ T\ v; R(v); F(v))$$

holds whenever every element of type T that satisfies the range-restriction predicate R, also satisfies formula$F$. Similarly, an existentially quantified formula of the form

$$(\texttt{\textbackslash exists}\ T\ v; R(v); F(v))$$

holds whenever there is some element of type T that satisfies the range-restriction predicate R, that also satisfies formula F.

JML provides declarative expressions that are quite useful in writing specifications. The `\reach(x,T,f)` expression denotes the smallest set containing the object denoted by `x`, if any, and all objects accessible through field `f` of type `T`. If `x` is null, then this set is empty.

Quantifiers and reach expressions are difficult to analyze using tools based on theorem provers. This is because having quantifiers makes the specification logic undecidable, i.e., there is no algorithm that can determine for an arbitrary formula whether the formula holds or not. Similarly, since `reach` (a reflexive-transitive closure operator) cannot be defined in classical first-order logic, no complete characterization of this operator can be made by using a classical first-order logic theorem prover.

`\reach` expressions evaluate to objects of class `JMLObjectSet`. This class belongs to the so called *model* classes. Model classes are introduced in JML to represent mathematical constructions like sets, bags, integers and reals. In the above invariant, `has` and `int_size` are methods defined within class `JMLObjectSet`, testing membership and returning the number of elements stored respectively.

Listing 5.3. A JML represents clause

```
1     //@ model non_null JMLObjectSequence seq;
2
3     /*@ represents seq \such_that
4       @  (size == seq.int_size()) &&
5       @  (head == null ==> seq.isEmpty()) &&
6       @  (head != null ==>
7       @        (head==seq.get(0) && tail==seq.get(size-1))) &&
8       @  (\forall int i ; i >= 0 && i < size - 1;
9       @        ((Node)seq.get(i)).next == seq.get(i + 1));
10      @*/
```

A model field is introduced for modelling purposes, and therefore is not part of the implementation. Model (or abstract) fields are defined in JML to describe an ideal, not concrete, state. Model fields definitions are not completed unless a `represents` clause is provided. The purpose of this class is to link the abstract field value to the actual concrete structure.

Once again, a model class is referred. This time, class `JMLObjectSequence`. The JML specification for this class denotes a mathematical sequence. Fields `int_size`, `isEmpty` and `get` return the sequence size, if the sequence has no stored elements, and i-th sequence element respectively. The `represents` clauses states that the field `seq` is equal to the sequence of all `Node` objects stored contained in the `LinkList` structure.

Listing 5.4. Computing the set of updated locations

```
1     /*@ normal_behavior
2       @    requires index >= 0 && index < seq.int_size();
3       @    ensures \result == seq.get(\old(index));
4       @ also
5       @ exceptional_behavior
6       @    requires index < 0 || index >= seq.int_size();
7       @    signals_only IndexOutOfBoundsException;
8       @*/
```

```
9       /*@ pure @*/ /*@ nullable @*/ Node getNode(int index) {
10          ...
11      }
```

In Listing 5.4 a contract for method `get` is provided. The `nullable` annotation indicates that the method may return *null* values. The modifier `pure` states that under no circumstances this method may have side-effects.

The `normal_behavior` and `exceptional_behavior` clauses specify what happens if a normal and abnormal termination occurs respectively. A method ends normally if no exception is signalled. On the contrary, a method ends abnormally if an exception is thrown. Notice that in the latter case the specification predicates on `result`, stating a condition over the method's return value. As in the former specification case the method returns no value, a clause `signals_only` is used to predicate over the Throwable objects the method may signal. In the above exceptional specification case, the method is constrained to throw only instances of class `IndexOutOfBoundsException`.

The `requires` and `ensures` clauses are used to denote the method precondition (what clients must comply) and the method postcondition (what clients may assume) for every kind of behaviour. Another common JML clause is `assignable`, which allows to define what side-effects the method may have.

The `\result` expression can be used in the ensures clause of a non-void return type method. It refers to the value returned by the method. Given an expression e, the expression `\old(e)` denotes the value of expression *e*, evaluated in the pre-state.

5.2 Translating JML to DynJML

From the previous description of the JML language, is it easy to see that JML and DynJML are very close both syntactically and semantically. Nevertheless, any translation from JML to DynJML requires solving many impedances between these two languages. We will discuss a semantic preserving translation in the rest of this section.

Initial transformations

As Java expressions may not be side-effect free, a transformation is applied to the Java source code. This phase introduces temporal variables in order to make all Java expressions side-effect free prior to the actual translation. This process of transformation will be illustrated by example in Table 5.1.

Initial source code	Transformed source code

`return (getAnInt(parseInt(aString))==2);`	`int t1;`
	`int t2;`
	`t1 = parseInt(aString);`
	`t2 = getAnInt(t1);`
	`return (t2==2);`

Table 5.1. Transforming Java expressions

Notice that, nested invocations are also not valid DynJML expressions, the previous transformation appropriately replaces this kind of expressions with simpler statements preserving the Java source's initial behaviour.

In the same manner, as Java delays the computation of conditionals until its result is needed (namely, eager evaluation), a second transformation is performed to replace complex conditionals with a series of nested conditionals each composed by only one expression to be evaluated. Transforming a logical disjunction is presented in Table 5.2.

Translation to DynJML

Once the expressions are transformed into side-effect free form, translating a Java program into a DynJML program happens to follow quite straightforwardly.

The translation begins by mapping each class and interface in the Java type hierarchy to a new signature in DynJML. Similarly, JML model fields are also represented with DynJML fields. Non static fields are mapped to DynJML fields. Static fields are handled as fields from a distinguished singleton signature named *StaticFields*. The only purpose of this signature (which do not have a Java counterpart) is to store all static fields defined in the type hierarchy. Notice that the translation of the Java class hierarchy is straightforward due to the signature extension mechanism provided by DynJML.

Each `invariant` clause is transformed into a corresponding DynJML *object_invariant*. Analogously, `represents` clauses are translated.

As overloading is supported by DynJML, no special action is taken for overload method. On the other hand, if a Java method is overridden in any subclass, the corresponding DynJML program is annotated as *virtual*.

Most JML expressions may be directly encoded as DynJML expressions. Quantified expressions are mapped as Alloy's quantifications due to its equivalent se-

Initial source code	Transformed source code

`if ( A || B ) {`
`  do something ...`
`}`
	`boolean t1;`
	`if (A) {`
	`  t1 = true;`
	`} else {`
	`  if (B) {`
	`  t1 = true;`
	`  } else {`
	`  t1 = false;`
	`  }`
	`}`
	
	`if (t1==true) {`
	`  do something...`
	`}`

Table 5.2. Transforming Java conditionals

mantic. Occurrences of `\result` expression are substituted by the DynJML variable *result*. In DynAlloy, state variables to be evaluated in the post-state are primmed, and therefore the translation of an expression `\old(e)` solely consists on removing primes from expressions. Given an expression e, a type T and a field f, `\reach(e,T,f)` denotes the set of objects of type T reachable from (the object denoted by) e by traversing field f. Since Alloy provides the reflexive closure operator, we translate:

$$\texttt{\textbackslash reach}(e, T, f) \mapsto (\texttt{e.*f \& T})$$

The `non_null` modifier deserves a special comment. All fields that were specified as *non null* fields are assumed to be non null at the program entry. Similarly, they are asserted to be non null at the program finalization. These conditions are added to the original program specification.

JML behaviour cases are encoded as DynJML specification cases. The `pure` annotation is represented in DynJML with a fresh specification case of the form:

Listing 5.5. A purity specification case

```
1 SpecCase {
```

```
2     requires { true }
3     modifies { NOTHING }
4     ensures { true }
5  }
```

JML `assignable` clauses are mapped as DynJML *modifies* clauses. Location descriptors `\nothing` and `\everything` are directly mapped to *NOTHING* and *EVERYTHING* respectively.

Exceptions and JML Behaviours

DynJML supports no constructs for raising and handling exceptions. In order to encode the possibility of an exceptional return value, a new output parameter named *throws* is added to each procedure during translation. The intended meaning for this parameter is to store the throw exception object in case the program reaches an abnormal terminal. Notice that the value of the *result* argument must be ignored in this scenario.

When a JML `normal_behaviour` clause is translated, the condition `throws'==null` is added to the *ensures* clause within the resulting DynJML specification case. This coerces the implicit JML condition for normal behaviour cases that the program must finish normally.

On the contrary, since JML `exceptional_behaviour` clauses describe scenarios where the execution must end abnormally, the condition `throws'!=null` is added. Moreover, the `signals` and `signals_only` clauses are traslated respectively as follows:

signals (E ex) R(ex)	$\mapsto$	(throws' in E) implies R(throws')
signals_only E1,...,En	$\mapsto$	throws' in E1+...+En

In order to avoid executing a statement if an exception has occurred, while translating to DynJML each statement is guarded with a DynJML *assertion* stating that the actual value for variable `throw` is null. If this is not the case, the statement can not be executed because the normal control-flow was interrupted. A special care is taken for Java structures that explicitly deal with exception handling such as `try-catch`.

Runtime exceptions are those exceptions that can be thrown during the normal operation of the Java Virtual Machine. For example, a `NullPointerException` object is thrown when a null dereference occurs. We handle null dereference in DynJML by guarding any field access of the form `E.f` by adding the statements shown in Listing 5.6 before the actual dereference.

Listing 5.6. Representing null pointer exceptions

```
1 if no E or E==null {
2   createObject<NullPointerException>[throw]
3 }
```

This secures that, if expression E is *null*, a fresh NullPointerException object is allocated and stored in `throw`, interrupting the normal control-flow.

JDK classes

The *java.lang* package provides several classes that are fundamental to the design of the Java programming language such as :

- `Object`: the root of the Java class hierarchy
- *wrappers* for primitive values (Integer,Boolean,Character,etc.)
- Runtime exceptions (`NullPointerException`, `IndexOutOfBoundsExceptions`,etc.)

Clearly not fundamental (although useful), the *java.util* package provides collections including sets, maps, and lists. DynJML provides an specification for the following classes from *java.lang*:

Object	Thrownable
Integer	Exception
Boolean	RuntimeException
Byte	NullPointerException
String	IndexOutOfBoundsException
Class	ClassCastException

DynJML also provides specifications for collections. Since sets, maps and lists are legal Alloy entities, the DynJML programs operate over this abstract Alloy domain instead of the actual Java implementation for collection classes. Although some precision is lost due to this encoding, the abstraction leads to a better analysis.

JML model classes such as `JMLObjectSet` and `JMLObjectSequence` using the same abstractions.

5.3 Bounded Verification

As with JForge and Miniatur, the bounded verification analysis takes as input a JML annotated Java program, and a scope on the analysis. It searches within the scope of analysis for a trace that violates the JML contract.

The scope of analysis must constrain :

- a number of times to unroll each loop if no loop invariant was written
- a number of times to unfold each recursive call
- a bitwidth to limit the range of primitive integer values
- a cardinality for each JML class

While the first two limits are meant to bound the length of program traces under scrutiny, the last two bound the state-space search. As explained in Dennis' PhD. dissertation [26], all these limits results in under-approximation, eliminating (but never adding) possible behaviours. Because of this, if a counterexample is found during the analysis, this counterexample can not be spurious. The counterexample found serves as a witness of the contract violation. If no counterexample is found, since the analysis exhaustively searches within the state-space, it can be concluded that the specification is valid within the scope of analysis. Nevertheless, the specification may not be valid since a counterexample could be found if a greater scope of analysis were searched.

The bounded verification begins by translating the JML input program into DynJML. This is later translated into DynAlloy, and finally to Alloy. The scope of analysis for limiting traces is used during the translation from DynAlloy to Alloy. The scope for bounding the state-space is converted into a appropriate scope of analysis at Alloy level. The Alloy Analyser is then invoked using the Alloy model and the appropriate Alloy command using the bound. If Alloy finds a counterexample, it is subsequently lifted back to JML.

We will discuss in the following chapters some techniques to improve the bounded verification of JML programs.

5.4 Implementation Details

In this section we mention some implementations details. Not only the source code for Alloy is publicly available. All the necessary software and tools required in order to generate the source code are freely available, too.

TACO's current implementation depends on a variety of open-source projects: Alloy, ANTLR, Apache ANT, Apache Commons, JUnit, Log4J, JML, MultiJava, and KodKod

JML and MultiJava are required in order to successfully parse the input JML annotated Java source files. The resulting data is then appropriately translated into DynJML.

As some DynJML are previously defined (such as the JDK classes), an DynJML grammar parser is used to read the DynJML models. These models are later transformed into DynAlloy models, which are later translated into Alloy.

Chapter 6

A New Predicate for Symmetry Breaking

The process of SAT-based analysis relies on an implicit traversal of the space of plausible models (i.e., those that satisfy the specification) while looking for a model that does not satisfy the property being checked. As mentioned before, if this procedure finds one such model we know that a counterexample of the property exists. A model in this context is a valuation of the propositional variables. Thus, the size of the search space is exponential on the number of propositional variables, and we should strive to reduce its size.

Permutations of signature atoms (also called *symmetries*) do not alter the truth value of Alloy formulas. Therefore, once a valuation μ is considered, those valuations originated from μ by permuting atoms should be avoided. One way to do this is by introducing *symmetry breaking predicates* ruling out certain models. For instance, Alloy includes general purpose symmetry breaking predicates [81].

In this chapter we present symmetry breaking predicates tailored to avoid permutations in the Alloy representation of the Java memory heap.

6.1 SAT-based symmetry breaking

Let us consider the following Java classes for implementing singly-linked structures:

```
public class List {
  LNode head;
}

public class LNode {
```

```
  LNode next;
  Integer key;
}
```

For the above Java classes, the resulting Alloy model includes the signature definitions shown below:

```
one sig null {}

sig List {
  head : LNode + null
}

sig LNode {
  next : LNode + null,
  key : Integer + null
}

sig Integer {}
```

The following Alloy predicate describes acyclic lists that do not store *null* values:

```
pred acyclic_non_null[l :  List] {
  all n :  LNode |
    n in l.head.*next
      implies n !in n.^next and n.key!=null
}
```

Running the predicate in the Alloy Analyzer using the command

```
run acyclic_non_null for exactly 1 List,
                exactly 4 LNode,
                exactly 1 Integer
```

yields (among others) the instances shown in Fig. 6.1. Notice that the lists instance in the right-hand side is a permutation (on signature LNode) of the other one. This shows that while the symmetry breaking predicates included in Alloy remove many symmetries, some still remain. As breaking all symmetries would require, in the general case, the construction of an exponentially large symmetry breaking predicate, Alloy only constructs a small, polynomially-sized predicate that breaks some symmetries. This is usually enough to get good performance, but it does not break all symmetries.

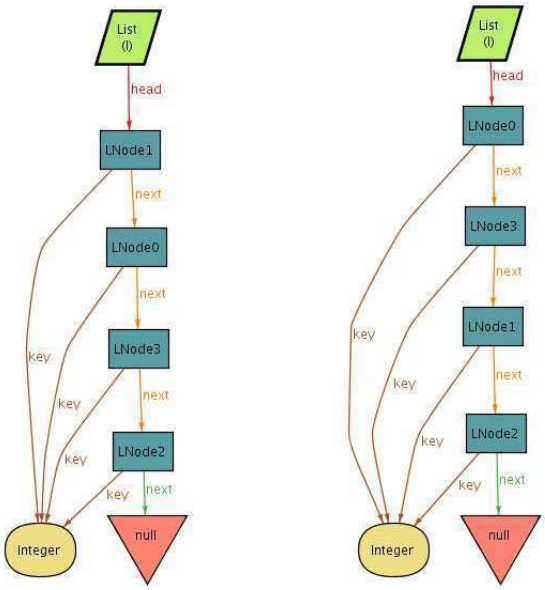


Figure 6.1. Two isomorphic list instances found by Alloy Analyzer

The ability to reduce the state space is central to scalability. Pruning the state space by removing permutations on signature LNode contributes to improving the analysis time by orders of magnitude.

Revisiting the singly linked lists example previously shown, it is easy to see that a predicate forcing nodes to be traversed in the order LNode0 $\rightarrow$ LNode1 $\rightarrow$ LNode2 $\rightarrow \ldots$ removes all symmetries.

6.2 An algorithm for generating symmetry breaking axioms

In this section we present a novel family of predicates that canonicalize arbitrary heaps.

Our model of Java heaps consists of graphs $\langle N, E, L, R\rangle$ where N (the set of heap nodes), is a set comprising elements from signature Object and appropriate value signatures (int, String, etc.). E is the set of edges, contains pairs $\langle n_1, n_2\rangle \in N \times N$. L is the edge labeling function. It assigns Java field names to edges. An edge between nodes n_1 and n_2 labelled f_i means that $n_1.f_i = n_2$. The typing must be respected. R is the root node labelling function. It assigns the receiver variable *this*, method arguments and static class fields to nodes. For example, a node n labelled *this* means that in the heap representation, the receiver object is node n.

for each type T **do**
  $k \leftarrow scope(T)$
  "one sig T_1,...,T_k *extends* T $\{\}$*"*
end for

for each recursive $r : T \mapsto$ (one T + null) **do**
  Add new field $fr : T \mapsto$ lone(T + null)
  Add new field $br : T \mapsto$ lone(T)
  Replace each "r" usage with expression "$fr + br$"
  Remove field r
  Add new axiom:
  "fact $\{$
    no (fr*.univ & br.univ) and*
    $T = fr.univ + br.univ$
  $\}$*"*
end for

Figure 6.2. The *instrument_Alloy()* procedure

The algorithm depends on defining an enumeration function for types, fields and heap root elements. The enumeration follows the order in which these elements appear in the Java source files. For the remainder of this subsection we will refer to $\{T_i\}_{i \in types}$, $\{f_i\}_{i \in fields}$ and $\{g_i\}_{i \in roots}$ as the ordered sets for types, fields and root nodes, respectively.

Instrumenting the Alloy model

In order to include the predicates we will instrument the Alloy model obtained by the translation from the annotated source code.

Besides the sets of ordered types, fields and root nodes, it is required to provide the finite scope of analysis for each type in order to instantiate the axioms and their auxiliary functions.

Let us denote by $scope(T)$ the function which returns an integer for each type T denoting the maximum number of atoms of type T a memory heap may have. The procedure *instrument_Alloy()* (shown in Figure 6.2) starts by introducing a singleton atom denoting each element of type T within the scope of analysis.

Once the singletons have been introduced, the procedure continue by splitting every *recursive* field. A field is considered *recursive* if domain and codomain (minus the *null* value) match. For instance, field next: LNode $\mapsto$ LNode+null is

considered a recursive field.

Each recursive field r from signature t is split into two partial functions (thus the `lone` modifier): fr the forward part of the field, mapping nodes to strictly greater nodes or `null`) and br the backward part of the field, mapping nodes to lesser nodes. Non-recursive fields are not modified. As Java fields must be total functions, the procedure also adds new facts stating that for each recursive field r, the domains of fr and br form a partition of r's domain, making $fr + br$ a well–defined total function.

The new fields obtained (that substitute the original ones) are meant to split the behaviour of the original fields between "forward" arcs and "backwards" arcs. Forward arcs may only map nodes to *greater* nodes (in terms of the element index) or null, while backwards arcs go to nodes that are *smaller* or equal in the ordering (and cannot go to null). Notice that forward arcs cannot lead to a cycle.

Because of the presented instrumentation, the set of original Alloy fields is partitioned into forward fields, backward fields, and non-recursive fields.

The instrumentation also modifies the facts, functions, predicates and assertions of the original model by replacing each occurrence of a recursive field r_i with the expression $fr_i + br_i$.

The auxiliary functions

The procedures shown in this subsection allow us to introduce the necessary auxiliary functions prior to introducing the symmetry breaking axioms.

Procedure *local_ordering()* (shown in Figure 6.3) generates auxiliary functions for:

- establishing a linear order between elements of type T (function *next_T*).
- returning the least object (according to the ordering *next_T*) in an input subset (function *min_T*).
- returning the nodes in signature T smaller than the input parameter (function *prevs_T*).

Notice that all these functions are constrained to operations among the elements of type T. We will consider them as "local" ordering auxiliary functions.

On the other hand, procedure *global_ordering()* (shown in Figure 6.4) is intended to provide functions which operate on all heap elements. This procedure defines Alloy functions for:

- establishing a linear order between elements of all types (function *global-Next*)

for each type T **do**
  $k \leftarrow scope(T)$
  "*fun nextT[] : T* $\rightarrow$ *lone T* $\{$
    $\langle T_1, T_2\rangle + \langle T_2, T_3\rangle + ... + \langle T_{k-1}, T_k\rangle$
  $\}$
  fun minT[os: set T] : lone T $\{$
    os - os.^nextT[]
  $\}$
  fun prevsT[o: T] : set T $\{$
    o.^(~ nextT[])
  $\}$"
end for

Figure 6.3. The *local_ordering()* **procedure**

- returning the least object (according to the ordering *globalNext*) in an input subset (function *globalMin*).

We will consider a node n' to be a parent of n if there exists a non-recursive field or a forward field f such that $n'.f = n$. A node may have no parents (in case it is a root node), or have several parent nodes. In the latter case, among the parents we will distinguish the minimal one (according to a global ordering) and will call it the *min parent* of n. The procedure *define_min_parent()* (shown in Figure 6.5) defines a min parent function for each type T. If n belongs to type T, $minP_T[n]$ returns the min parent of n (if any).

Notice that in the definition of function $minP_T$ we are only considering forward fields and non-recursive fields with target type T.

Key to the symmetry breaking predicates we are introducing is the notion of *reachable* objects. We consider a heap node to be *reachable* if it may be accessed during the program execution by traversing the memory heap.

Procedure *define_freach()* (presented in Figure 6.6) defines a function `FReach` denoting all objects that may be reachable by accessing either non-recursive fields or forward fields. This definition is a more economic (regarding the translation to a propositional formula) description of the reachable heap objects since no mention to the backward fields is needed.

The symmetry breaking predicates

The rest of the algorithm outputs axioms to canonicalize the order in which heap nodes are traversed. Intuitively, we will canonicalize heaps by ordering nodes

"fun globalNext[] : Object → lone Object {"
for each type T_i **do**
  if $i > 0$ **then**
    "+"
  end if
  "$\langle T_{i_1}, T_{i_2} \rangle + \langle T_{i_2}, T_{i_3} \rangle$ +...+ $\langle T_{i_{k-1}}, T_{i_k} \rangle$"
  if T_{i+1} exists **then**
    $max_{T_i} \leftarrow$ maximum singleton from T_i
    $min_{T_{i+1}} \leftarrow$ minimum singleton from T_{i+1}
    "+ $\langle max_{T_i}, min_{T_{i+1}} \rangle$ "
  end if
end for
"}
fun globalMin[s: set Object] : lone Object {
  s - s.^globalNext[]
}"

Figure 6.4. The *global_ordering()* **procedure**

for each type T **do**
  Let $f_1,\ldots,f_i$ be all non-recursive fields targeting T
  Let $r_1,\ldots,r_j$ be all recursive fields targeting T
  Let $g_1,\ldots,g_k$ be all root nodes of type T
  "fun $minP_T$[o : T] : Object {
  o !in $(g_1 + \cdots + g_k)$
    => globalMin[$(f_1 + \ldots + f_i + fr_1 + \ldots + fr_j)$.o]
    else none }"
end for

Figure 6.5. The *define_min_parent()* **procedure**

Let $f_1,\ldots,f_i$ be all non-recursive fields
Let $r_1,\ldots,r_j$ be all recursive fields
Let $g_1,\ldots,g_k$ be all root nodes
"fun FReach[] : set Object {
  $(g_1+\ldots+g_k)$.$(f_1+...+f_k+fr_1+...+fr_n)$ - null*
}"

Figure 6.6. The *define_freach()* **procedure**

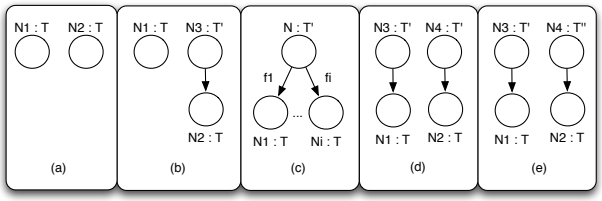


Figure 6.7. Comparing nodes using their min-parents.

looking at their parents in the heap. We will explain the rest of the algorithm by considering the possibilities depicted in Fig. 6.7. Given two nodes of type T, we distinguish the following cases :
(a) Both nodes are root nodes.
(b) One node is a root node and the other is a non-root node.
(c) Both nodes are non-root nodes with the same min-parent.
(d) Both nodes are non-root nodes with different min-parents of the same type T'.
(e) Both nodes are non-root nodes with min-parents of different types.

Notice that any pair of nodes of type T is included in one (and only one) of these cases.

Procedure *order_root_nodes()* (presented in Figure 6.8) outputs an axiom that sorts two root nodes of type T. The axiom forces every pair of root nodes to obey the ordering in which formal parameters and static fields (namely, the root nodes) were declared in the source Java file.

Procedure *root_is_minimum()* (presented in Figure 6.9) creates an axiom that constrains the first non-null root node of type T to store the minimum element. The conjunction of this axiom and the one generated by the procedure *order_root_references()* (Figure 6.8) forces root nodes to always be smaller than non root nodes.

Procedure *order_same_min_parent()* (shown in Figure 6.10) outputs an axiom which sorts nodes $N_1, \ldots, N_i$ of the same type such that $\texttt{minP}[N_1] = \ldots = \texttt{minP}[N_i] = N$. Notice that since Java fields are functions, there must be i different fields $f_1, \ldots, f_i$ such that $N.f_1 = N_1$, $N.f_2 = N_2$, etc. We then use the ordering in which the fields were declared in the source Java file to sort $N_1, \ldots, N_i$.

Procedure *order_same_min_parent_type()* (presented in Figure 6.11) creates an axiom which sorts nodes with different min-parents belonging to the same type T'. Let N_1 (with min parent N_3) and N_2 (with min parent N_4) be nodes of the same type. If N_3 and N_4 are distinct and have the same type, then the axiom sorts N_1 and N_2 following the order between N_3 and N_4.

```
for each type T do
  "fact {"
  Let g_1, ..., g_k be the root references of type T
  for i = 0 to k do
    for j = i + 1 to k do
      "( g_i ≠null "
      for w = i + 1 to j − 1 do
        " and ( g_w=null "
        for v = 0 to i do
          " or g_w = g_v"
        end for
        " ) "
      end for
      " and g_j ≠null "
      for h = 0 to i do
        " and g_h ≠ g_j"
        " and g_h ≠ g_i"
      end for
      ") implies ⟨g_i, g_j⟩ ∈ nextT []"
    end for
  end for
  "}"
end for
```

Figure 6.8. The *order_root_nodes()* procedure

Finally, the procedure *order_diff_min_parent_types()* shown in Figure 6.12 sorts nodes N_1 and N_2 of type T whose min parents have different type. Notice that the axiom orders the nodes following the order in which the classes of the parent nodes were defined in the source Java file.

In order to avoid "holes" in the ordering, for each signature T procedure *avoid_holes()* (presented in Figure 6.13) adds a fact stating that whenever a node of type T is reachable in the heap, all the smaller ones in the ordering are also reachable.

Symmetry breaking predicates: An example

In order to make the introduction of the symmetry breaking predicates more amenable to the reader, we now present an example. Let us consider the red-

for each type T **do**
  "fact {"
  Let $g_1, \ldots, g_k$ be the *root references* of type T
  for $i = 1$ to k **do**
    "(("
    for $j = 1$ to $i - 1$ **do**
      "g_j=null and "
    end for
    $min_T \leftarrow$ minimum singleton from T
    "$g_i \neq$null) implies $g_i = min_T$)"
    if $i < k$ **then**
      " and "
    end if
  end for
  "}"
end for

Figure 6.9. The *root_is_minimum()* procedure

black trees whose class hierarchy is presented in Fig. 6.14.

The scopes for analysis will be:

- 1 RBTree atom,
- 5 RBTNode atoms, and
- 5 Integer atoms.

Following procedure *instrument_Alloy()* (Fig. 6.2), fields `left` and `right` are replaced with fields `fleft` (the forward part of `left` field, `bleft` (the backward part of `left`), `fright` (the forward part of `right`) and `bright` (the backward part of `right`), respectively. Only these two fields are split because these are the only fields that match the definition of *recursive*.

The procedure introduces the following axiom to force `fleft+bleft` to be a well–defined total function:

```
fact {
  no (fleft.univ & bleft.univ) and
  RBTNode = fleft.univ + bleft.univ
}
```

```
for each T, T′ types do
  Let f_1, ..., f_k be all non-recursive and forward fields of type T′ → T
  if k > 1 then
    "fact {
    all disj o1,o2: T |
      let p1=minP_T[o1] |
      let p2=minP_T[o2] |
      ( o1+o2 in FReach[] and
      some p1 and some p2 and
      p1=p2 and p1 in T′ ) implies ("
    for i = 1 to k − 1 do
      if i > 1 then
        "and"
      end if
      "( ( p1.f_i=o1 "
      for j = i + 1 to k do
        for l = i + 1 to j − 1 do
          "and minP_T[p1.f_l] ≠ p1"
        end for
        "and p1.f_j=o2 ) implies o2 = o1.nextT[] )"
      end for
    end for
    ")}"
  end if
end for
```

Figure 6.10. The *order_same_min_parent()* procedure

for each T, T' types **do**
  if exists a field $f{:}T \mapsto T'$ **then**
    "fact {
    all disj o1,o2: T |
      let p1=minP_T[o1] |
      let p2=minP_T[o2] |
      (o1+o2 in FReach[] and
      some p1 and some p2 and
      p1!=p2 and p1+p2 in T′ and p1 in prevsT′[p2])
        implies o1 in prevsT[o2]
    }"
  end if
end for

Figure 6.11. The *order_same_min_parent_type()* **procedure**

for each T type **do**
  Let $\{T'_i\}$ be the ordered subset of types
  s.t. exist fields $f : T'_j \mapsto T$, $g : T'_k \mapsto T$, $j < k$.
  "fact {
  all disj o1,o2: T |
    let p1=minP_T[o1] |
    let p2=minP_T[o2] |
    (o1+o2 in FReach[] and
    some p1 and some p2 and
    p1 in T'_j and p2 in T'_k)
      implies o1 in prevsT[o2])
  }"
end for

Figure 6.12. The *order_diff_min_parent_types()* **procedure**

for each T type **do**
  "fact {
  all o: T |
    o in FReach[] implies
      prevs_T[o] in FReach
  }"
end for

Figure 6.13. The *avoid_holes()* **procedure**

```
class RBTNode extends Object {
  boolean is_black;
  Integer value;
  RBTNode left;
  RBTNode right;
}

class RBTree extends Object {
  RBTNode root;
}
```

Figure 6.14. A red-black trees class hierarchy

A similar Alloy fact is appended in order to make `fright+bright` a total function.

Our model of Java heaps consists of graphs $\langle N, E, L, R \rangle$. In the present example, nodes are the objects from signatures `RBTree`, `RBTNode` and `Integer`, or the value `null`. Labels correspond to field names, and R is the receiver variable `this`, of type `RBTree`.

Let us assume that types appear in the Java source files in the following order:

1. RBTNode
2. RBTree
3. Integer

Also, assume that field declarations appear in the following order:

1. is_black : RBTNode $\mapsto$ one boolean
2. value : RBTNode $\mapsto$ one (Integer+null)
3. fleft : RBTNode $\mapsto$ lone (RBTNode+null)
4. bleft : RBTNode $\mapsto$ lone (RBTNode+null)
5. fright: RBTNode $\mapsto$ lone (RBTNode+null)
6. bright: RBTNode $\mapsto$ lone (RBTNode+null)
7. root : RBTree $\mapsto$ one (RBTNode+null)

From executing procedure *local_ordering()*, new auxiliary functions are introduced. For the example (only for signature `RBTNode`), the procedure outputs:

```
fun next_RBTNode[] : RBTNode -> lone RBTNode {
   RBTNode_0->RBTNode_1
   + RBTNode_1->RBTNode_2
   + RBTNode_2->RBTNode_3
   + RBTNode_3->RBTNode_4 }

fun min_RBTNode [os: set RBTNode] : lone RBTNode {
   os - os.^(next_RBTNode[]) }

fun prevs_RBTNode[o : RBTNode] : set RBTNode {
   o.^(~next_RBTNode[]) }
```

Similarly, the procedure outputs function definitions for types `RBTree` and `Integer`.

Procedure *global_ordering()* (Fig. 6.4) outputs the declaration of function `globalNext`. This function provides an ordering on all objects in the heap. As the reader may notice, each `next_T` is subsumed in `globalNext`.

```
fun globalNext[]: Object -> Object {
  RBTNode_0->RBTNode_1 + ... + RBTNode_3->RBTNode_4
  + RBTNode_4->RBTree_0
  + RBTree_0->Integer_0
  + Integer_0->Integer_1 + ... + Integer_3->Integer_4
}
```

The following min-parent functions are defined by procedure *define_min_parent()* (Fig. 6.5). Notice that since no fields have objects of type `RBTree` in their range, no `minP_RBTree` function is defined.

```
fun minP_RBTNode[o: RBTNode]: Object {
  globalMin[(fleft+fright+root).o]
}

fun minP_Integer[o: Integer]: Object {
  globalMin[(value).o]
}
```

Procedure *define_freach()* (Fig. 6.6) yields the definition of a function that characterizes the reachable heap objects:

```
fun FReach[]: set Object {
  this.*(value + fleft + fright + root)
}
```

Notice that field `is_black` is excluded because `boolean` values are not heap objects and the `FReach` function returns a set of heap objects. So far no axioms were introduced other than those constraining the additions of forward and backward fields to be total functions.

Procedure *order_root_nodes()* (Fig. 6.8) does not output any axioms because there is only one root node, namely, `this`, of type `RBTree`. Procedure *root_is_minimum()* (Fig. 6.9) outputs:

```
fact {
  this != null implies this = RBTree_0
}
```

Regarding procedure *order_same_min_parent()* (Fig. 6.10), since there is only one field from type `RBTree` to type `RBTNode`, there are no two objects with type `RBTNode` with the same min parent in signature `RBTree`. The same reasoning applies to `RBTNode` and `Integer`. Notice instead that there are two forward fields from type `RBTNode` to type `RBTNode` (namely, `fleft` and `fright`). The axiom produced by *order_same_min_parent()* (described below) orders objects of type `RBTNode` with the same min parent of type `RBTNode`:

```
fact {
 all disj o1, o2 : RBTNode |
  let p1 = minP_RBTNode[o1] |
  let p2 = minP_RBTNode[o2] |
   (o1+o2 in FReach[] and
   some p1 and some p2 and
   p1 = p2 and p1 in RBTNode ) implies
     (( o1 = p1.fleft and
     o2 = p1.fright) implies
        o2 = o1.next_RBTNode[])
}
```

Procedure *order_same_min_parent_type()* (Fig. 6.11) yields three axioms. The first one, included below, orders objects of type `RBTNode` with different min parents of type `RBTNode`. The other two axioms are similar and sort objects of type `Integer` with different `RBTNode` min parents, and objects of type `RBTNode` with different `RBTree` min parents. Notice that, since the scope of type `RBTree` is equal to 1, the last axiom is identically true and can be automatically removed.

```
fact {
 all disj o1, o2 : RBTNode |
  let p1 = minP_RBTNode[o1] |
```

```
  let p2 = minP_RBTNode[o2] |
   (o1+o2 in FReach and
   some p1 and some p2 and
   p1!=p2 and p1+p2 in RBTNode and
   p1 in prevs_RBTNode[p2])
       implies o1 in prevs_RBTNode[o2]
}
```

Only one type (`RBTNode`) satisfies the conditions required by procedure *order_diff_min_parent_types()* (Fig. 6.12). In effect, `RBTNode` is the only type for which there are fields pointing to it coming from two different types (for instance, fields `fleft` and `root` have the right typing). The procedure generates the following axiom, which orders objects of type `RBTNode` whose min parents are one of type `RBTree`, and the other of type `RBTNode`:

```
fact {
  all disj o1, o2 : RBTNode |
    let p1 = minP_RBTNode[o1] |
    let p2 = minP_RBTNode[o2] |
      (o1+o2 in FReach and
      some p1 and some p2 and
      p1 in RBTree and p2 in RBTNode)
           implies o1 in prevs_RBTNode[o2]
}
```

Procedure *avoid_holes()* (Fig. 6.13) outputs the following axiom for signature `RBTNode`:

```
fact {
  all o : RBTNode |
    o in FReach[] implies
      prevs_RBTNode[o] in FReach[]
}
```

This procedure also generates similar axioms for signatures `RBTree` and `Integer`. Notice that since *scope*(`RBTree`) = 1, the resulting fact is identically true and is automatically removed.

6.3 A Correctness proof

The following theorems show that the instrumentation is correct.

THEOREM 6.3.1 *Given a heap H for a model, there exists a heap H' isomorphic to H and whose ordering between nodes respects the instrumentation. Moreover, if an edge $\langle n_1, n_2\rangle$ is labeled r (with r a recursive field), then: if n_1 is smaller (according to the ordering) than n_2 (or n_2 is* `null`*), then $\langle n_1, n_2\rangle$ is labeled in H' fr. Otherwise, it is labeled br.*

Proof sketch: For each signature T, let $n_{root,T}$ be the number of root objects from T. For each pair of signatures T, T', let $n_{T,T'}$ be the number of objects from T whose min-parent has type T' (notice that although min-parent is not fully defined, we can determine its type due to the linear ordering imposed on signature names). Assign the first $n_{root,T}$ elements from T to root elements. Notice that this satisfies the condition depicted in Fig. 6.7(a). Use the linear ordering between types and assign, for each signature T', $n_{T,T'}$ objects from T for nodes with min-parent in T'. When doing so, assign smaller objects (w.r.t. the linear ordering `nextT`) to smaller (w.r.t. the linear ordering on signature names) T' signature names. Notice that this satisfies the conditions depicted in Fig. 6.7(b) and 6.7(e). It only remains to determine the order between nodes in the same type and whose min-parent has the same type. Follow the directions given in Fig. 6.7(b)–(d). This defines a bijection b between nodes in H and nodes in H'. We still have to label heap edges. Let n_1, n_2 be nodes in H connected via an edge labeled r. Notice that $b(n_1)$ and $b(n_2)$ have the same type as n_1 and n_2, respectively. Therefore, if r is not recursive, use r as the label for the edge between $b(n_1)$ and $b(n_2)$. If r is recursive, then n_1 and n_2 have the same type or $n_2 =$ `null`, and the same is true for $b(n_1)$ and $b(n_2)$. Thus, since there is a total order on each type, if $b(n_1) < b(n_2)$ or $n_2 =$ `null` set the label of the edge between $b(n_1)$ and $b(n_2)$ to fr. Otherwise, set it to br. ∎

Theorem 6.3.1 shows that the instrumentation does not miss any bugs during code analysis. If a counterexample for a partial correctness assertion exists, then there is another counterexample that also satisfies the instrumentation.

THEOREM 6.3.2 *Let H, H' be heaps for an instrumented model. If H is isomorphic to H', then $H = H'$.*

Proof sketch: Suppose $H \neq H'$. Since they are isomorphic there must be a minimal position i_0 where their breadth-first search traversals differ. Let $n_1, n_2, \ldots, n_{i_0}, \ldots, n_i, \ldots$ be the traversal of H, and $m_1, m_2, \ldots, m_{i_0}, \ldots, m_i, \ldots$ be the traversals of H'. Since i_0 is minimal, it must be $n_1 = m_1$, $n_2 = m_2$, …, $n_{i_0-1} = m_{i_0-1}$. Moreover, let us assume without loss of generality that $n_{i_0} > m_{i_0}$. Let $j > i_0$ such that

$$n_{i_0} > n_j \text{ and } m_{i_0} < m_j \,. \tag{6.1}$$

Such j exists because by axiom `compactT` there are no holes in the traversals. Note that n_{i_0} and n_j have the same type. Also, $minP[n_{i_0}] = minP[m_{i_0}]$. We now use the axioms in order to reach to contradictions. Let us discuss the case in which $minP[n_{i_0}] = minP[n_j]$, with the other cases being similar. Since H and H' are isomorphic, $minP[m_{i_0}] = minP[m_j]$. But, according to `orderTNodeCondition(c)`, the order between n_{i_0} and n_j must be the same as the order between m_{i_0} and m_j (contradicting (6.1)). ■

Theorem 6.3.2 shows that the instrumentation indeed yields a canonicalization of the heap.

Chapter 7

Parallel Computation of Tight Bounds

In this chapter we describe a novel technique to reduce the number of propositional variables during analysis. This technique relies on the symmetry breaking predicates we have introduced previously.

The Alloy language has a relational semantics. This means that in order to translate an Alloy specification to a SAT problem, the technique focuses on the translation of fields as relations. Given scopes s for signature `S` and t for signature `T`, one can determine the number of propositional variables required in order to represent a field `f : S -> one (T+null)` in the SAT model. Notice that `S` and `T` will contain atoms $S_1, \ldots, S_s$ and $T_1, \ldots, T_t$, respectively. Alloy uses a matrix $M_{\mathtt{f}}$ holding $s \times (t+1)$ propositional variables to represent the field `f` (see Fig. 7.1).

Intuitively, a variable p_{S_i,T_j} $(1 \leq i \leq s, 1 \leq j \leq t)$ models whether the pair of atoms/identifiers $\langle S_i, T_j \rangle$ belongs to `f` or, equivalently, whether $S_i.\mathtt{f} = T_j$. A variable $p_{S_i,\mathtt{null}}$ models whether $S_i.\mathtt{f} = \mathtt{null}$. Actually, as shown in Fig. 5.1, Alloy models are not directly translated to a SAT problem, but to the intermediate

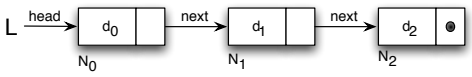
$M_{\mathtt{f}}$	T_1	T_2	$\ldots$	T_t	null
S_1	p_{S_1,T_1}	p_{S_1,T_2}	$\cdots$	p_{S_1,T_t}	$p_{S_1,\mathtt{null}}$
S_2	p_{S_2,T_1}	p_{S_2,T_2}	$\cdots$	p_{S_2,T_t}	$p_{S_2,\mathtt{null}}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
S_s	p_{S_s,T_1}	p_{S_s,T_2}	$\cdots$	p_{S_s,T_t}	$p_{S_s,\mathtt{null}}$

Figure 7.1. Matrix representation of an Alloy field.

language KodKod [81].

A distinguishing feature of Alloy's backend, KodKod, is that it enables the prescription of partial instances in models. Indeed, each Alloy 4 field `f` is translated to a matrix of propositional variables as described in Fig. 7.1, together with two *bounds* (relation instances) $L_{\texttt{f}}$ (the lower bound) and $U_{\texttt{f}}$ (the upper bound). As we will see, these bounds provide useful information. Consider for instance relation `next` from the singly-linked list model presented in the previous chapter. If a tuple $\langle N_i, N_j \rangle \notin U_{\texttt{next}}$, then no instance of field `next` can contain $\langle N_i, N_j \rangle$, allowing us to replace p_{N_i,N_j} in $M_{\texttt{next}}$ (the matrix of propositional variables associated with relation `next`) by the truth value *false*. Similarly, if $\langle N_i, N_j \rangle \in L_{\texttt{next}}$, pair $\langle N_i, N_j \rangle$ must be part of any instance of field `next` (allowing us to replace variable p_{N_i,N_j} by the truth value *true*). Thus, the presence of bounds allows us to determine the value of some entries in the KodKod representation of a given Java field.

Assume that the class invariant for representing a singly linked list requires lists to be acyclic. Assume also that nodes have identifiers $N_0, N_1, N_2, \ldots$. Thus, a list instance will have the shape



Notice that since lists are assumed to be acyclic, it is easy to see that some tuples are deemed to never be contained in any `next` relation instance. Since no node may refer to itself, there is no instance such that either of tuples $\langle N_0, N_0 \rangle$, $\langle N_1, N_1 \rangle$ and $\langle N_2, N_2 \rangle$ is contained in relation `next`. If we could determine this before translating to a propositional formula, then these tuples could be safely removed from the U_{next} upper bound. By doing so, propositional variables representing membership of these tuples (namely, p_{N_0,N_0}, p_{N_1,N_1} and p_{N_2,N_2}) could be replaced with value *false*, leading to a formula with fewer variables. Since, in the worst case, the SAT-solving process grows exponentially with respect to the number of propositional variables, getting rid of variables often improves (as we will show in Sec. 8) the analysis time significantly. In our example, determining that a pair of atoms $\langle N_i, N_j \rangle$ can be removed from the bound $U_{\texttt{next}}$ allows us to remove a propositional variable in the translation process. When a tuple is removed from an upper bound, the resulting bound is said to be *tighter* than before. In this section we concentrate on how to determine if a given pair can be removed from an upper bound relation and therefore improving the analysis performance.

Up to this point we have made reference to three different kinds of bounds, namely:

- The bounds on the size of data domains used by the Alloy Analyzer. Generally, these are referred to as *scopes* and should not be confused with the intended use of the word *bounds* in this section.

- In DynAlloy, besides imposing scopes on data domains as in Alloy, we bound the number of loop unrolls. Again, this bound is not to be confused with the notion of bound that we will use in this section.

- In this section we made reference to the lower and upper bounds ($L_{\mathtt{f}}$ and $U_{\mathtt{f}}$) attached to an Alloy field `f` during its translation to a KodKod model. For the remaining of this section, we use the term *bound* to refer to the upper bound $U_{\mathtt{f}}$.

Complex linked data structures usually have complex invariants that impose constraints on the topology of data and on the values that can be stored. For instance a class invariant for the red-black tree structure we introduced in Sec. 6.2 states that:

1. For each node n in the tree, the keys stored in nodes in the left subtree of n are always smaller than the key stored in n. Similarly, keys stored in nodes in the right subtree are always greater than the key stored in n.

2. Nodes are colored red or black, and the tree root is colored black.

3. In any path starting from the root node there are no two consecutive red nodes.

4. Every path from the root to a leaf node has the same number of black nodes.

In the Alloy model result of the translation, Java fields are mapped to total functional relations. For instance, field *left* is mapped to a total functional relation. Suppose that we are interested in enumerating instances of red black trees that satisfy a particular predicate. This predicate could be the above representation invariant, or a method precondition involving red black trees. Let us assume it is the above invariant. Furthermore, let us assume that:

1. nodes come from a linearly ordered set, and

2. trees have their node identifiers chosen in a canonical way (for instance, a breadth-first order traversal of the tree yields an ordered listing of the node identifiers).

In particular, these assumptions may be fulfilled by using the symmetry breaking predicates introduced in Sec. 6.2. Following the breadth-first order heap canonization, given a tree composed of nodes $N_0, N_1, \ldots, N_k$, node N_0 is the tree root, $N_0.\mathit{left} = N_1$, $N_0.\mathit{right} = N_2$, etc. Observe that the breadth-first ordering allow us to impose more constraints on the structure. For instance, it is no longer possible that $N_0.\mathit{left} = N_2$. Moreover, if there is a node to the left of node N_0, it *has* to be node N_1 (otherwise the breadth-first listing of nodes would be broken). At the Alloy level, this means that $\langle N_0, N_2 \rangle \in \texttt{left}$ is infeasible, and the same is true for $N_3, \ldots, N_k$ instead of N_2. Recalling the discussion at the beginning of this section, this means that we can get rid of several propositional variables in the translation of the Alloy encoding of the invariant to a propositional SAT problem. Actually, as we will show in Sec. 8, for a scope of 10 red-black tree nodes, this analysis allows us to reduce the number of propositional variables from 650 to 200.

The usefulness of the previous reasonings strongly depends on the following two requirements:

1. being able to guarantee, fully automatically, that nodes are placed in the heap in a canonical way, and

2. being able to automatically determine, for each class field `f`, what are the infeasible pairs of values that can be removed from the bound $U_{\texttt{f}}$.

To cope with requirement 1 we will rely on the symmetry breaking predicates we introduced in Sec. 6.2. With respect to requirement 2, in Sec. 7.1 we will present a fully automatic and effective technique for checking feasibility.

7.1 Symmetry Breaking and Tight Bounds

In the previous section we discussed the representation of red-black trees. While in the original Alloy model functions *left* and *right* are each encoded using $n \times (n + 1)$ propositional variables, due to the canonical ordering of nodes and to the class invariant we can remove arcs from relations. In order to determine whether edges $N_i \rightarrow N_j$ can be part of field `F` or can be removed from $U_{\texttt{F}}$, TACO proceeds as follows:

1. Synthesizes the instrumented model following the procedure shown in Sec. 6.2.

2. Adds to the model the class invariant as an axiom.

3. For each pair of object identifiers N_i, N_j, it performs the following analysis:

```
pred NiToNjInF[] {
  Ni+Nj in FReach[] and Ni->Nj in F
}
run NiToNjInF for scopes
```

In the example, for field `fleft` we must check, for instance,

```
pred TNode0ToTNode1Infleft[] {
   TNode0 + TNode1 in FReach[] and
   TNode0->TNode1 in fleft
}
run TNode0ToTNode1Infleft for exactly 1 Tree,
                              exactly 5 TNode,
                              exactly 5 Data
```

If a "run" produces no instance, then there is no memory heap in which N_i->N_j `in F` that satisfies the class invariant. Therefore, the edge is infeasible within the provided scope. It is then removed from $U_{\mathtt{F}}$, the upper bound relation associated to field F in the KodKod model. This produces tighter KodKod bounds which, when the KodKod model is translated to a propositional formula, yield a SAT problem involving fewer variables.

All these analysis are independent. A naive algorithm to determine feasibility consists of performing all the checks in parallel. This algorithm is presented in Fig. 7.2. Unfortunately, the time required for each one of these analysis is highly irregular. Some of the checks take milliseconds, and others may exhaust available resources while searching for the complex instances that have to be produced.

7.2 An iterative algorithm for bound computation

The algorithm for bound refinement we used in Galeotti et al. [40] (whose pseudocode is given in Fig. 7.3), is an iterative procedure that receives a collection of Alloy models to be analyzed, one for each edge whose feasibility must be checked. It also receives as input a threshold time T to be used as a time bound for the analysis. All the models are analyzed in parallel using the available resources. Those individual checks that exceed the time bound T are stopped and left for the next iteration. Each analysis that finishes as unsatisfiable tells us that

```
function fill_queue(upper_bounds, spec):  int
 int task_count=0
 For each edge A->B in upper_bounds Do
  M := create_Alloy_model(A->B,upper_bounds,spec)
  task_count++
  ENQUEUE(<A->B,M>, workQ)
 End For
 return task_count


function NAIVE_MASTER(scope, spec):  upper_bounds
 // Master creates and stores tasks into shared queue
 workQ := CREATE_QUEUE()
 upper_bounds := initial_upper_bounds(spec, scope)
 task_count := fill_queue(upper_bounds, spec)

 // Master receives results from slaves
 result_count := 0
 While result_count!=tasks_count Do
  <A->B,analysis_result> := RECV()
  result_count++
  If analysis_result==UNSAT Then
   upper_bounds:= upper_bounds - A->B
 End While
 return upper_bounds


function NAIVE_SLAVE()
 While !isEmpty(workQ) Do
  <A->B,M> := DEQUEUE()
  analysis_result := run_Alloy(M)
  SEND(master,<A->B,analysis_result>)
 End While
```

Figure 7.2. The naive parallel algorithm for bound refinement.

an edge may be removed from the current bound. Satisfiable checks tell us that the edge cannot be removed. After all the models have been analyzed, we are left with a partition of the current set of edge models into three sets: unsatisfiable checks, satisfiable checks, and stopped checks for which we do not have a conclusive answer. We then refine the bounds (using the information from the unsatisfiable models) for the models whose checks were stopped. The formerly stopped models are sent again for analysis giving rise to the next iteration. This process, after a number of iterations, converges to a (possibly empty) set of models that cannot be checked (even using the refined bounds) within the threshold T. Then, the bounds refinement process finishes. Notice that in TACO's algorithm the most complex analysis (those reaching the timeout), get to use tighter bounds in each iteration.

The following theorem shows that the bound refinement process is safe, i.e., it does not miss faults.

THEOREM 7.2.1 *Let H be a memory heap exposing a fault. Then there exists a memory heap H' exposing the bug that satisfies the instrumentation and such that for each field* `g`*, the set of edges with label* `g` *(or* `bg` *or* `fg` *in case* `g` *is recursive) is contained in the refined $U_{\mathtt{g}}$.*

Proof sketch: Let H' be the heap from Theorem 6.3.1. It satisfies the instrumentation and, since H' is isomorphic to H, it also exposes the fault. Assume there is in H' an edge $N_i \rightarrow N_j$ labeled `g`, such that $N_i \rightarrow N_j \notin U_{\mathtt{g}}$. Since during code analysis TACO includes the class invariant as part of the precondition, heap H' must satisfy the invariant. But since $N_i \rightarrow N_j \notin U_{\mathtt{g}}$, the Alloy analysis

```
pred NiToNjInF[] {
  Ni in FReach[] and
  Ni->Nj in F
}
run NiToNjInF for scopes
```

must have returned UNSAT. Then, there is no memory heap that satisfies the invariant and contains the edge $N_i \rightarrow N_j$, leading to a contradiction. ∎

For most of the case studies we report in Sec. 8 it was possible to check all edges using this algorithm. Since bounds only depend on the class invariant, the signatures scopes and the typing of the method under analysis, the same bound is used (as will be seen in Sec. 8) to improve the analysis of different methods. By extending TACO's architecture, once a bound is computed, it is stored in a bounds repository as shown in Fig. 7.4.

7.3 An eager algorithm for bound computation

It is generally the case that the number of processors is significantly smaller than the number of analysis that can be run in parallel. As we already mentioned, analysis time for feasibility checks is highly irregular. Thus, by the time an analysis is allocated to a given processor, verdicts from previous edges may have

been already reported. In the TACO algorithm presented in Fig. 7.3 a *generational* approach is taken. This means that although an UNSAT verdict is known for a given edge, this information has no effect before the current iteration is finished.

An alternative approach for computing bounds is to make use of UNSAT information as soon as it is available. This leads to a third algorithm, shown in Fig. 7.5. For the rest of this article, we will refer to this alternative algorithm as the *eager* algorithm. The main characteristic of this algorithm is that upper bounds are updated as soon as an UNSAT certificate is obtained. Therefore, Alloy models being allocated for analysis make use of the most recent upper bound information. Also, since the Alloy Analyzer outputs a model whenever a feasibility check returns SAT, the algorithm marks all variables corresponding to edges that are reachable from the root nodes in that model, as satisfiable. This improves the efficiency of the tool by avoiding the analysis of those edges.

```
global TIMEOUT

function fill_queue(upper_bounds, spec):  int
 int task_count=0
 For each edge A->f B in upper_bounds Do
  M := create_Alloy_model(A->f B,upper_bounds,spec)
  task_count++
  ENQUEUE(<A->f B,M>, workQ)
 End For
 return task_count

function ITERATIVE_MASTER(scope, spec):  upper_bounds
 workQ := CREATE_QUEUE()
 upper_bounds := initial_upper_bounds(spec, scope)
 Do
  task_count := fill_queue(upper_bounds, spec)
  result_count := 0
  timeout_count := 0
  unsat_count := 0
  While result_count != tasks_count Do
   <A->f B, analysis_result> := RECV()
   result_count++
   If analysis_result==UNSAT Then
    upper_bounds := upper_bounds - A->f B
   Else If analysis_result==TIMEOUT Then
    timeout_count ++
  End While
  If unsat_count==0
   return upper_bounds
 Until timeout_count==0
 return upper_bounds

function ITERATIVE_SLAVE()
 While workQ>0 Do
  <A->f B,M> := DEQUEUE()
  analysis_result := run_stoppable_Alloy(M, TIMEOUT)
  SEND(master, <A->f B, analysis_result>)
 End While
```

Figure 7.3. TACO's algorithm for iterative bound refinement.

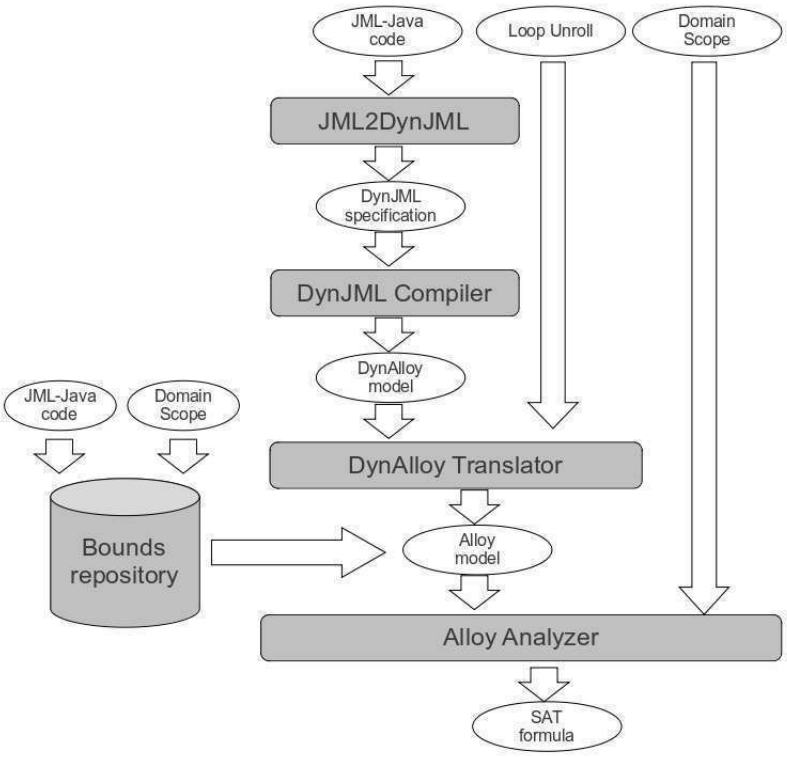


Figure 7.4. TACO architecture extended with a bounds repository.

```
global TIMEOUT, upper_bounds, edgeQ

function EAGER_MASTER():
 edgeQ := CREATE_QUEUE()
 upper_bounds := initial_upper_bounds(spec, scope)
 For each A->f B in upper_bounds Do
  ENQUEUE(edgeQ,A->f B)

function EAGER_SLAVE():
 While edgeQ>0 Do
  <A->f B,M> := DEQUEUE(edgeQ)
  M := create_Alloy_Model(A->f B, upper_bounds, spec)
  <analysis_result, I> := run_stoppable_Alloy(M,TIMEOUT)
  If analysis_result==UNSAT Then
   upper_bounds := upper_bounds - A->f B
  Else If analysis_result==SAT Then
   For each A'->f' B' in I Do
    remove(edgeQ, A'->f' B')
  Else
   ENQUEUE(A->f B, edgeQ)
  End If
 End While
```

Figure 7.5. TACO's algorithm for eager bound refinement.

Chapter 8

Evaluation

In this chapter we report the results obtained from conducting several experiments. We analyze 7 collection classes with increasingly complex class invariants. Using these classes we will study the performance of TACO in several ways. We will denote by TACO$^-$ the translation implemented in TACO, but without using neither the symmetry reduction axioms nor the tight bounds. In Sec. 8.2 we compare the effect on analysis time of the inclusion of the symmetry breaking predicates. This is achieved by comparing TACO$^-$ with TACO. In Sec. 8.3, we compare the parallel algorithms for computing bounds presented in Figs. 7.3 and 7.5. Section 8.4 reports on the impact of using tighter bounds. Finally, in Secs. 8.5 and 8.6, we compare TACO with several tools in two settings. The first one is a comparison with JForge [28] (a state-of-the-art SAT-based analysis tool developed at MIT). Since the classes we analyze are correct, this allows us to compare the tools in a situation where the state space must be exhausted. The second one is when we study the error-finding capabilities of TACO against several state-of-the-art tools based on SAT-solving, model checking and SMT-solving.

8.1 Experimental Setup

In this section we analyze methods from collection classes with increasingly rich invariants. We will consider the following classes:

- **LList:** An implementation of sequences based on singly linked lists.
- **AList:** The implementation `AbstractLinkedList` of interface `List` from the Apache package `commons.collections`, based on circular doubly-linked lists.

- **CList:** A caching circular double linked list implementation of interface `List` from the Apache package `commons.collections`.
- **BSTree:** A binary search tree implementation from Visser et al. [86]
- **TreeSet:** The implementation of class `TreeSet` from package `java.util`, based on red-black trees.
- **AVLTree:** An implementation of AVL trees obtained from the case study used in Belt et al. [9].
- **BHeap:** An implementation of binomial heaps used as part of a benchmark in Visser et al. [86].

In all cases we are checking that the invariants are preserved. Also, for classes `LList`, `AList` and `CList`, we show that methods indeed implement the sequence operations. Similarly, in classes `TreeSet`, `AvlTree` and `BSTree` we also show that methods correctly implement the corresponding set operations. For class `BHeap` we also show that methods correctly implement the corresponding priority queue operations. We also analyze a method for extracting the minimum element from a binomial heap, that contains a previously unknown fault (we discus it extensively in Sec. 8.6).

Loops are unrolled up to 10 times, and no contracts for called methods are being used (we inline their code). We set the scope for signature `Data` equal to the scope for nodes. We have set a timeout (TO) of 10 hours for each one of the analysis. Entries "OofM" mean "out of memory error". When reporting times using TACO, we are not adding the times (given in Table 8.3) to compute the bounds. Still, adding these times does not yield a TO for any of the analysis that did not exceed 10 hours.

The parallel algorithms for computing bounds were run in a cluster of 16 identical quad-core PCs (64 cores total), each featuring two Intel Dual Core Xeon processors running at 2.67 GHz, with 2 MB (per core) of L2 cache and 2 GB (per machine) of main memory. Non-parallel analysis, such as those performed with TACO *after* the bounds were computed, or when using other tools, were run on a single node. The cluster OS was Debian's "etch" flavor of GNU/Linux (kernel 2.6.18-6). The message-passing middleware was version 1.1.1 of MPICH2, Argonne National Laboratory's portable, open-source implementation of the MPI-2 Standard. All times are reported in `mm:ss` format. Those experiments for which there exists a non-deterministic component in the behavior of the algorithm were run ten times and the value reported corresponds to the average of all execution times.

8.2 Analyses Using Symmetry Breaking Predicates

As mentioned before, none of the main contributions of this dissertation were implemented in TACO$^-$İn this sense, the analysis time of TACO$^-$ can be used as a reference value for measuring the improvement produced by the inclusion of symmetry breaking predicates as well as by the use of tight bounds.

In Table 8.1 we compare the analysis time of TACO$^-$ against a version of TACO that only adds the symmetry breaking predicates (we will call this intermediate version TACO^{sym}). In other words, bounds are neither computed nor used by TACO^{sym}. The cell highlighting denotes which tool needed the smaller amount of computing time. If both tools required the same amount of computing time or both tools reached the time limit, no cell was highlighted.

Table 8.2 shows the improvement of using the symmetry breaking predicates discussed in Section 6.2. All methods under analysis are correct with respect to their specification (except method ExtractMin from class binomial heap, that contains a fault we discovered). The first column shows the maximum scope for which TACO$^-$ achieves the analysis within the time threshold of 10 hours. Similarly, the second column shows the same information for TACO^{sym}. The third and fourth columns show the analysis times for TACO$^-$ and TACO^{sym} for those particular scopes. Finally, the last column shows the ratio between the time required by TACO^{sym} and TACO$^-$Às in Table 8.1, we distinguish the tool that reached the greater scope of analysis as well as the one that consumed the smaller time of analysis by highlighting the corresponding cells.

Observe that in most cases TACO^{sym} outperforms TACO$^-$ both in maximum scope for which the analysis ends within the time limit and in the amount of time spent in analysis for the maximum scope for which both tools finish. This can be seen in the fifth column corresponding to the analysis times ratio. To summarize the information of the table, 94% of cases show an increase of the maximum scope of analysis, while only for 1 case (6%) this value decreases. This was calculated on the basis of those cases were at least one of the tools reached the timeout limit. Considering all the experiments in the benchmark, TACO^{sym} increases the scope of analysis in 7.31 nodes, on average. When comparing the greater common scope for which both tools accomplish the analysis within the time limit, in 80% of the experiments a dramatic decrease in the time of analysis is shown. When calculating over these cases, the time required by TACO^{sym} to accomplish the analysis is, on average, 2.58% of the time consumed by TACO$^-$

			5	7	10	12	15	17	20
LList	contains	T^-	00:03	00:05	00:08	00:11	00:13	00:22	00:34
		T^s	00:00	00:01	00:05	00:09	00:25	00:46	00:50
	remove	T^-	00:04	00:09	01:14	00:33	04:26	01:25	02:57
		T^s	00:01	00:01	00:02	00:03	00:07	00:22	00:38
	insert	T^-	00:05	00:27	TO	TO	TO	TO	TO
		T^s	00:01	00:01	00:06	00:10	00:29	00:39	01:46
AList	contains	T^-	00:05	00:11	00:29	00:38	00:42	01:37	01:21
		T^s	00:01	00:04	00:32	00:45	02:22	07:46	243:54
	remove	T^-	00:04	00:05	01:02	26:22	TO	TO	TO
		T^s	00:01	00:01	00:02	00:05	01:00	04:49	258:21
	insert	T^-	00:06	00:14	11:25	347:39	TO	TO	TO
		T^s	00:01	00:03	00:16	00:38	03:21	15:08	TO
CList	contains	T^-	00:46	03:51	00:22	01:01	01:30	06:39	01:09
		T^s	00:01	00:06	00:25	01:48	04:50	18:18	TO
	remove	T^-	00:11	22:22	TO	TO	TO	TO	TO
		T^s	00:01	00:01	00:04	00:18	03:06	12:17	TO
	insert	T^-	02:43	TO	TO	TO	TO	TO	TO
		T^s	00:06	00:29	02:29	06:52	31:48	112:25	TO
BSTree	contains	T^-	16:30	320:39	TO	TO	TO	TO	TO
		T^s	00:03	00:50	136:15	TO	TO	TO	TO
	remove	T^-	02:07	TO	TO	TO	TO	TO	TO
		T^s	00:02	00:35	54:42	TO	TO	TO	TO
	insert	T^-	TO	TO	TO	TO	TO	TO	TO
		T^s	09:26	128:52	TO	TO	TO	TO	TO
TreeSet	contains	T^-	02:13	276:49	TO	TO	TO	TO	TO
		T^s	00:02	00:16	05:35	22:00	186:17	TO	TO
	remove	T^-	21:38	TO	TO	TO	TO	TO	TO
		T^s	01:05	10:48	TO	TO	TO	TO	TO
AVL	find	T^-	00:14	27:06	TO	TO	TO	TO	TO
		T^s	00:01	00:01	00:14	01:27	06:24	74:58	TO
	findMax	T^-	00:02	00:04	46:12	TO	TO	TO	TO
		T^s	00:02	00:17	03:30	11:25	177:17	TO	TO
	insert	T^-	01:20	335:51	TO	TO	TO	TO	TO
		T^s	00:11	01:05	16:51	64:01	TO	TO	TO
BHeap	findMin	T^-	00:12	11:41	TO	TO	TO	TO	TO
		T^s	00:01	00:04	00:20	00:52	05:15	14:32	45:47
	decKey	T^-	05:36	TO	TO	TO	TO	TO	TO
		T^s	00:11	01:30	34:36	120:39	TO	TO	TO
	insert	T^-	22:46	391:10	TO	TO	TO	TO	TO
		T^s	00:40	02:23	31:15	91:07	250:09	TO	TO

Table 8.1. Comparison of code analysis times for 10 loop unrolls using TACO$^-$ (T^-) and TACO^{sym} (T^s).

8.3 Computing Tight Bounds

In Chapter 7 we emphasized the fact that our technique allowed us to remove variables in the translation to a propositional formula. Each of the reported classes includes some field definitions. For each field `f` in a given class, dur-

		max T^- scope	max T^s scope	T^- time	T^s time	T^s time / T^- time
LList	contains	20	20	00:34	00:51	150%
	remove	20	20	02:17	00:38	21.47%
	insert	8	20	07:24	07:34	102.25%
AList	contains	20	20	01:20	243:54	18292%
	remove	13	20	150:57	00:08	0.09%
	insert	13	19	283:08	00:38	0.22%
CList	contains	20	19	00:39	116:40	17948%
	remove	9	19	123:07	00:04	0.05%
	insert	7	18	148:20	00:26	0.29%
BSTree	contains	7	11	320:39	00:49	0.25%
	remove	6	11	325:12	00:06	0.03%
	insert	3	7	02:38	00:08	5.06%
AvlTree	find	11	18	472:33	00:33	0.12%
	findMax	7	15	21:00	00:16	0.127%
	insert	7	14	245:35	01:05	0.44%
TrSet	contains	7	16	127:09	00:16	0.21%
	remove	6	7	412:11	04:51	1.19%
BHeap	findMin	8	20	202:54	00:05	0.04%
	decKey	7	12	560:34	01:29	0.26%
	insert	7	15	70:53	02:23	0.36%

Table 8.2. Comparison of code analysis times for 10 loop unrolls using TACO$^-$ (T^-) and TACO^{sym} (T^s).

ing the translation from Alloy to KodKod an upper bound $U_{\mathtt{f}}$ is readily built. We will call the union of the upper bounds over all fields, the *upper bound*. In Table 8.3 we report, for each class, the following:

1. The number of variables used by TACO$^-$ in the upper bound (#UB). That is, the size of the upper bound without using the techniques described in this article.

2. The size of the tight upper bound (#TUB) used by TACO. The tight upper bound is obtained by applying the bound refinement algorithm from Section 7.1 starting from the initial upper bound. Given a field f, we call initial upper bound to the instance of U_f in which all tuples belongs to U_f. The time required to build the initial upper bound is negligible.

3. The time required by the algorithm in Fig. 7.3 to build the tight upper

bound.

4. The time required by the algorithm in Fig. 7.5 to build the same tight upper bound.

Again, we distinguish the algorithm that consumed the smaller amount of time by highlighting the corresponding cell. For both algorithms, the initial timeout used during bound refinement for the individual analysis was set to 2'.

Table 8.3 shows that, in average, over 70% of the variables in the bounds can be removed. Let us now compare the performance of computing a tight bound by using the iterative algorithm (Fig. 7.3) and the eager algorithm (Fig. 7.5). Observe that, on average, a speed-up of approximately 1.95x is achieved by using the eager algorithm instead of the iterative algorithm for computing bounds. Both iterative and eager algorithms exceeded the 10 hour barrier for only one experiment (cyclic linked list and cache linked list respectively, both for a scope of 20).

Although the aforementioned savings are indeed significant, it is worth mentioning that they fail to achieve a major improvement in asymptotic terms. Figures 8.1 and 8.2 are introduced as two representative cases of the comparison of both algorithms. As these figures illustrate, projections of the same data on a logarithmic scale on the y-axis reveal some interesting offset shifts, yet hardly any impact on the slopes.

Both techniques suffer from a high number of aborted partial analysis. We are currently developing strategies to mitigate this problem. We hope that this will help us in devising a more scalable algorithm for computing tight bounds.

8.4 Analysing the Impact of Using Bounds

In this section we will show the results of systematically tightening the bounds to determine the effects of such change in the SAT-solver behaviour. Our hypothesis is that most times a tighter bound leads to a smaller analysis time.

In order to study the effect of tightening the bound, we run the same analysis varying only this parameter. Up to this point, we have referred to two kinds of bounds: the initial bound (all tuples) and the tightest bound (computed by the distributed algorithms). To evaluate the impact of using bounds we built several approximations ranging from the initial bound to the tightest bound. We produce a bound $B_{n\%}$ by removing those edges whose feasibility check was reported as UNSAT, and fall within the $n\%$ of the less expensive checks in term of analysis time. Given two edges e_1 and e_2, we say that e_1 is less expensive than e_2 if the time needed for obtaining a verdict for the feasibility of e_1 is less than that of

#Node		5	7	10	12	15	17	20
LList	#UB	30	56	110	156	240	306	420
	#TUB	9	13	19	23	29	33	39
	Time I	00:11	00:14	00:23	00:36	01:01	01:23	02:25
	Time E	00:11	00:11	00:15	00:24	00:47	01:04	01:37
AList	#UB	76	128	252	344	512	676	904
	#TUB	33	47	68	82	103	117	138
	Time I	00:16	00:25	00:51	01:26	02:47	09:28	TO
	Time E	00:11	00:14	00:51	00:33	00:55	03:15	300:20
CList	#UB	328	384	498	594	768	904	1138
	#TUB	97	127	172	210	240	277	322
	Time I	00:57	01:13	01:45	02:25	05:27	21:31	575:00
	Time E	00:35	00:46	01:11	01:38	01:46	05:16	TO
BSTree	#UB	90	168	330	468	720	918	1260
	#TUB	54	97	184	257	389	492	669
	Time I	00:22	00:34	01:04	01:46	03:19	05:32	25:10
	Time E	00:11	00:11	00:16	00:38	01:56	04:05	21:19
TrSet	#UB	170	280	650	852	1200	2006	2540
	#TUB	59	107	200	279	424	533	720
	Time I	00:49	01:13	03:03	05:11	11:30	44:23	97:04
	Time E	00:16	00:30	01:44	02:51	05:19	16:42	40:37
AVL	#UB	150	280	650	852	1200	2006	2540
	#TUB	55	98	177	251	389	491	669
	Time I	00:33	00:57	03:26	09:53	22:03	101:31	579:40
	Time E	00:17	00:32	01:55	03:46	10:36	47:25	168:23
BHeap	#UB	222	360	803	1053	1488	2394	2540
	#TUB	75	123	218	293	423	481	669
	Time I	00:44	01:12	04:00	06:48	20:13	62:50	211:20
	Time E	00:22	01:12	02:46	04:41	10:32	34:50	117:20

Table 8.3. Sizes for initial upper bounds (#UB) and for tight upper bounds (#TUB), and analysis time for computation of tight upper bounds using iterative algorithm (I) and eager algorithm (E).

e_2. Notice that, using this definition, the $B_{100\%}$ bound corresponds to the tightest bound, while the $B_{0\%}$ bound corresponds to the initial bound.

By using the stored logging information from running the distributed algo-

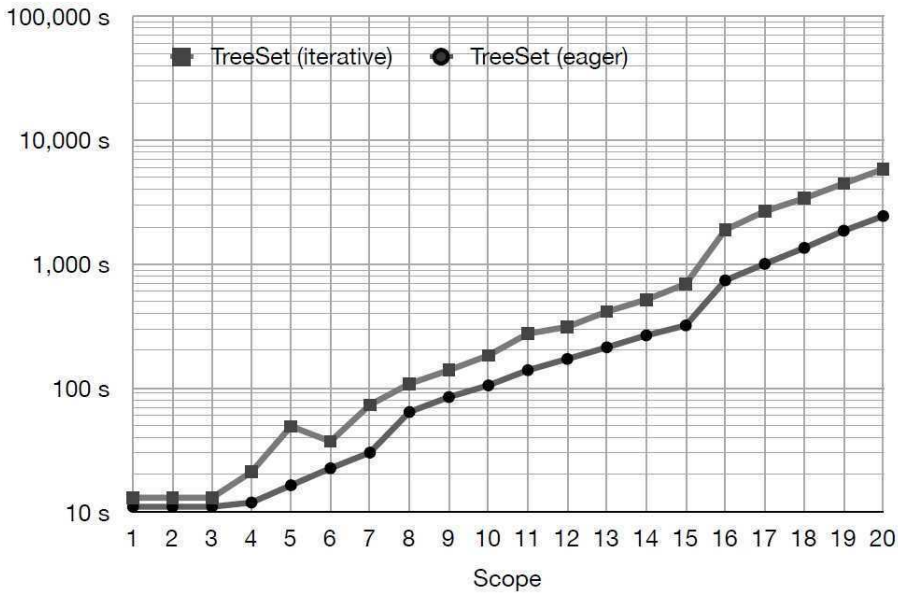


Figure 8.1. Analyses time (in logarithmic scale) for computing bounds of TreeSet using the iterative and the eager algorithms

rithm we built the following bounds: $B_{10\%}$, $B_{20\%}$, $B_{30\%}$, $B_{40\%}$, $B_{50\%}$, $B_{60\%}$, $B_{70\%}$, $B_{80\%}$, and $B_{90\%}$.

The reader may notice that computing bounds of different precision only makes sense when the iterative algorithm for computing bounds (Fig. 7.3) is used. This is because in the dynamic algorithm of Fig. 7.5 the analysis time for a given check is strongly influenced by the initial scheduling.

Once the bounds were defined for each collection class, we re-run each experiment varying the bound. The timeout was set again to 10 hours. We fixed the scope of each method under analysis to be the maximum value such that TACO (using any incremental bound) successfully completed the analysis within the time limit. The rationale behind this decision is examining the effect on the hardest problems.

Due to the small analysis times, the case studies corresponding to class `LList` were explicitly excluded from this assessment. For the remaining 17 methods under analysis, 7 exhibited a (almost) strictly monotonic decrease in the analysis time required as the bound gets tighter. The improvement is shown in logarithmic scale in Fig. 8.3.

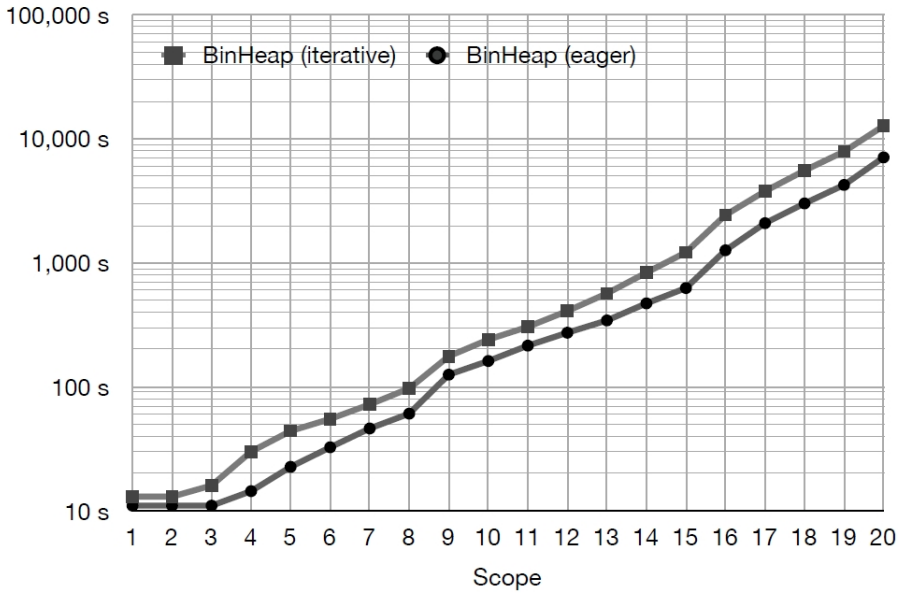


Figure 8.2. Analyses time (in logarithmic scale) for computing bounds of Binomial heap using the iterative and the eager algorithms

For the 7 methods under analysis shown in Fig. 8.4, a dramatic decrease in analysis time is also exhibited. Although some oscillations do occur for a couple of cases, the gain obtained from tightening the bound is clear.

Finally, for the 3 methods shown in Fig. 8.5 no improvement appears to be obtained by increasing the bound precision. These cases represent the 15% of all methods under analysis. On the contrary, the remaining 85% do exhibit an exponential improvement. Therefore, we conclude that the analysis of the selected benchmark is sensitive to tightening the bounds.

It is worth mentioning that, for those methods that do exhibit an improvement in the analysis as the bound precision grows, this improvement is also shown in smaller scopes. In order to illustrate the reader, we also report the results of the analysis times for the method *remove* for the AVL tree and method *insert* for the cached cyclic linked list. Figures 8.6 and 8.7 show as a gray-scale gradient the analysis time for both methods as the scope grows.

Figs. 8.6 and 8.7 show the relation between a tighter bound and the analysis time. It is easy to see that tightening the bound contributes in allowing the analysis to finish within the time limit for greater scopes.

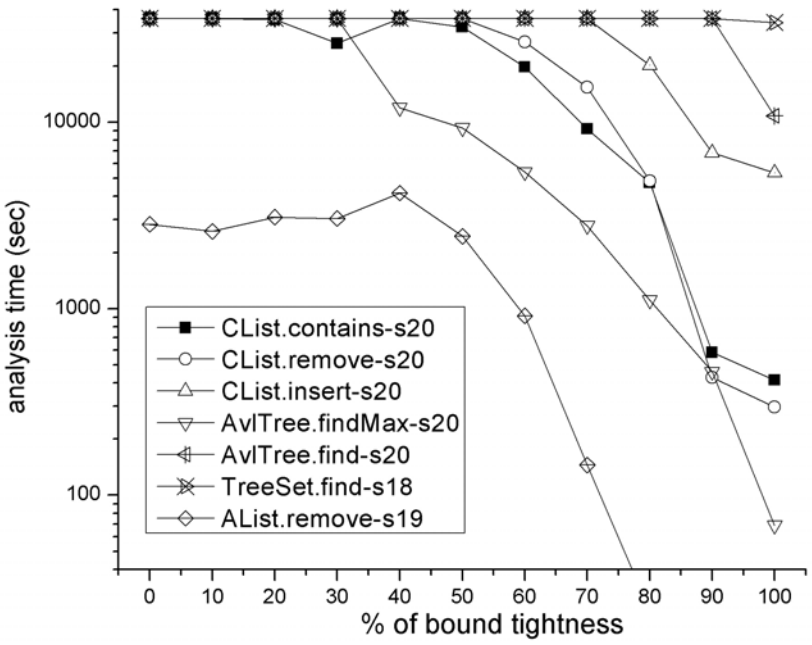


Figure 8.3. Analysis time as bound precision is increased.

8.5 Analysis of Bug-Free Code

In this section we present the results of comparing TACO with tight bounds with JForge, another SAT-based tool for Java code analysis. The results are shown in Table 8.4.

Table 8.4 shows that as the scope grows, in most cases (as the cell highlighting shows) TACO requires a smaller amount of time than JForge. While we will not present a detailed analysis of memory consumption, it is our experience that TACO uses less memory both during translation to a propositional formula as well as during SAT-solving, than JForge.

8.6 Bug Detection Using TACO

In this section we report on our experiments using TACO in order to detect faults, and will compare TACO to other tools. We will be analyzing method

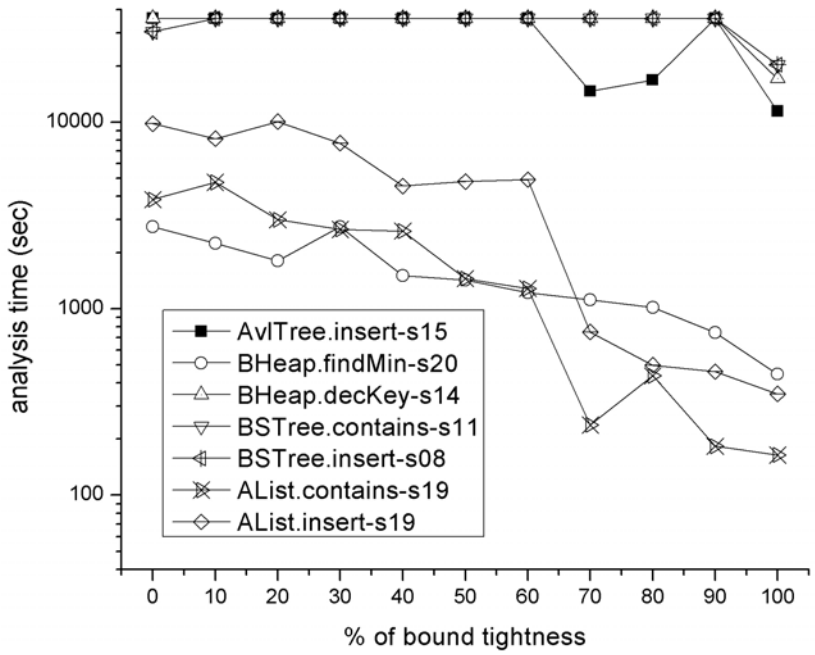


Figure 8.4. Analysis time as bound precision is increased.

`Remove` from classes `LList` and `CList`, and method `ExtractMin` from class `BHeap`. Due to the similarities in the analysis techniques, we will first compare TACO with TACO$^-$ and JForge, and later in the section we will also compare TACO with ESC/Java2 [13], Java PathFinder [85], and Kiasan [9]. We also used Jahob [10], which neither succeeded in verifying the provided specifications, nor provided an understandable counterexample (only raw formulas coming from the SMT-solvers).

Detecting Mutants

In order to compare JForge, TACO$^-$ and TACO we will generate mutants for the chosen methods using the muJava [63] mutant generator tool. After manually removing from the mutants set those mutants that either were equivalent to the original methods or only admitted infinite behaviors (the latter cannot be killed using these tools), we were left with 31 mutants for method `Remove` from class

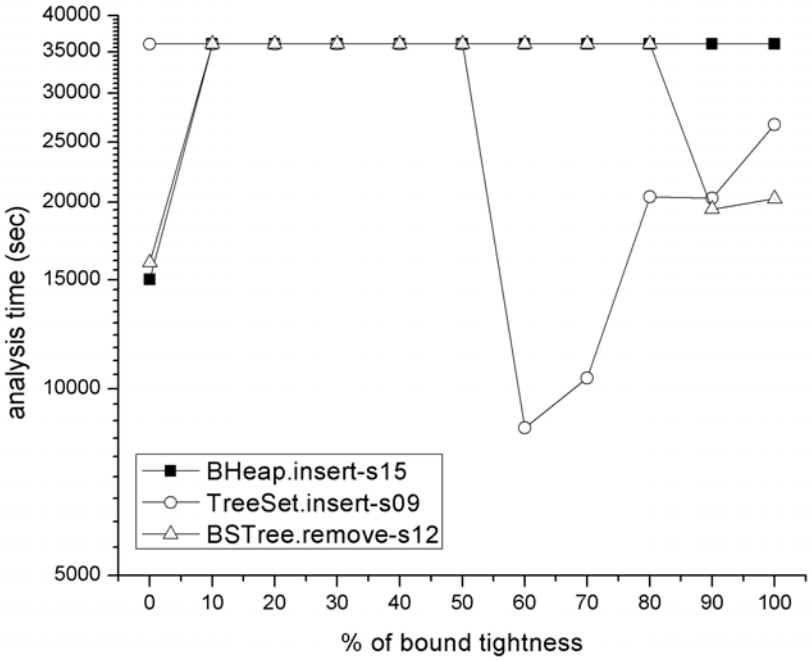


Figure 8.5. Analysis time as bound precision is increased.

`LList`, 81 mutants for method `Remove` from class `CList` and 50 mutants for method `ExtractMin` from class `BHeap`.

For all the examples in this section we have set the analysis timeout in 1 hour.

In Fig. 8.8 we report, for each method, the percentage of mutants that can be killed as the scope for the `Node` signature increases. We have set the scope for signature `Data` equal to the number of nodes. Notice that while the 3 tools behave well in class `LList`, TACO can kill strictly more mutants than TACO$^-$ and JForge the CList example. We can also see that as the scope grows, TACO$^-$ and JForge can kill fewer mutants. This is because some mutants that were killed in smaller scopes cannot be killed within 1 hour in a larger scope.

In order to report analysis times, we will carry out the following procedure, which we consider the most appropriate for these tools:

1. Try to kill each mutant using scope 1. Let T_1 be the sum of the analysis times using scope 1 for all mutants. Some mutants will be killed, while

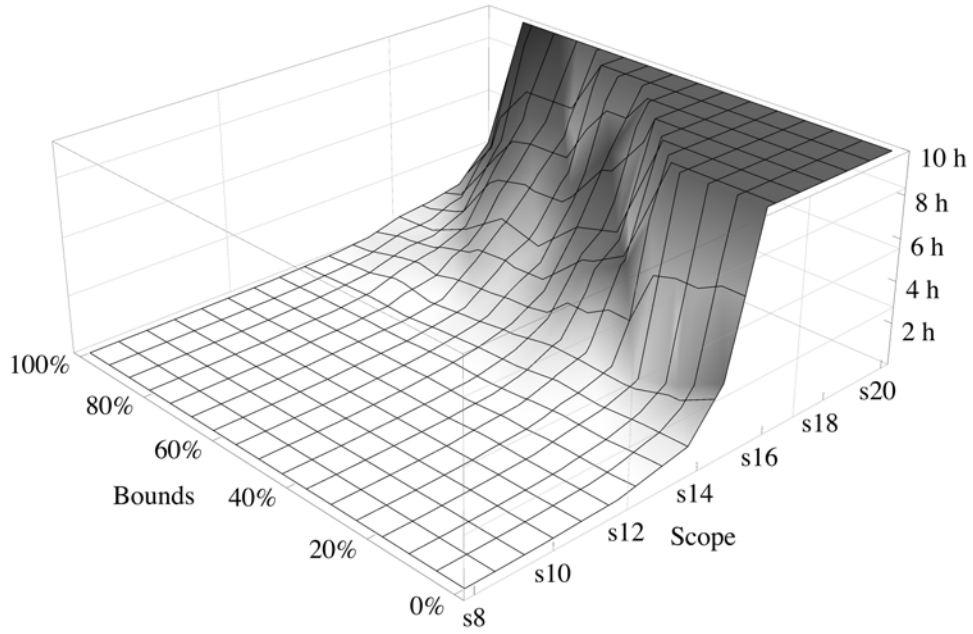


Figure 8.6. Analysis time for method *remove* of AvlTree as scope and bound tightness grows.

others will survive. For the latter, the analysis will either return UNSAT (no bug was found in that scope), or the 1 hour analysis timeout will be reached.

2. Take the mutants that survived in step 1, and try to kill them using scope 2. Let T_2 be the sum of the analysis times.

3. Since we know the minimum scope k for which all mutants can be killed (because TACO reached a 100% killing rate without any timeouts in scope k), repeat the process in step 2 until scope k is reached. Finally, let $T = \sum_{1 \leq i \leq k} T_i$.

Notice first that the previous procedure favors TACO$^-$ and JForge. In effect, if a tool is used in isolation we cannot set an accurate scope limit beforehand (it is the user's responsibility to set the limit). If a scope smaller than the necessary one is chosen, then killable mutants will survive. If a scope larger than the

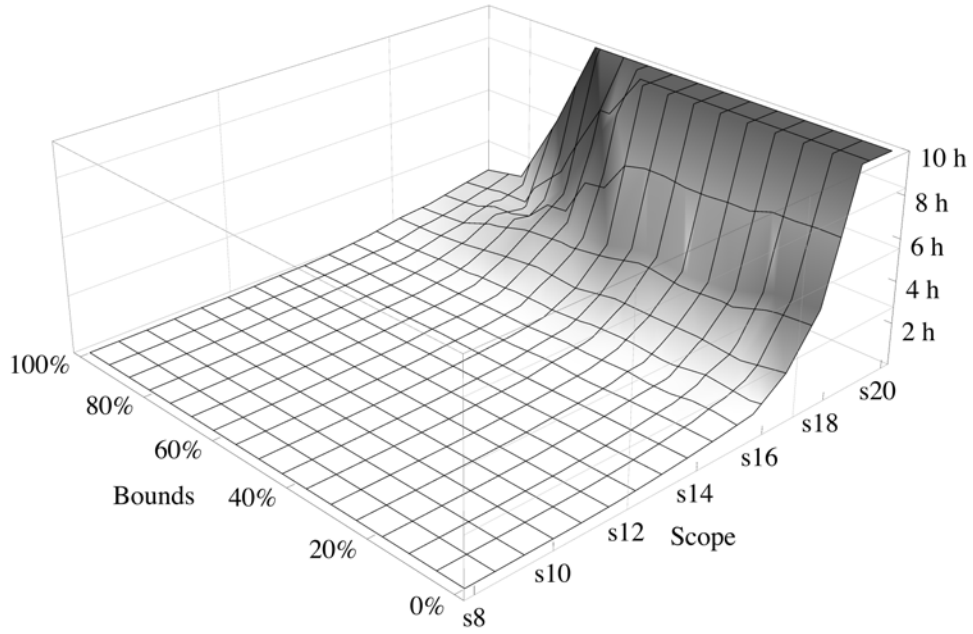


Figure 8.7. Analysis time for method *insert* of CList as scope and bound tightness grows.

appropriate one is set, then we will be adding 1 hour timeouts that will impact negatively on the reported times. Notice also that an analysis that reached the timeout for scope $i < k$ will be run again in scope $i + 1$. This is because we cannot anticipate if the timeout was due to a performance problem (the bug can be found using scope i but the tool failed to find the bug within 1 hour), or because the bug cannot be found using scope i. In the latter case it may happen that the mutant can be found in scope $i + 1$ before reaching the timeout. This is the situation in Table 8.4 for method `ExtractMin` where the timeout was reached by TACO for scope 12, yet the bug was found using scope 15 in less than 10 hours.

It is essential to notice that the same tight bound is used by TACO for killing all the mutants for a method within a given scope. Thus, when reporting analysis times for TACO in Table 8.5, we also add the time required to compute the bounds for scopes $1, \ldots, k$. In general we tried to use 10 loop unrolls in all cases. Unfortunately, JForge runs out of memory for more than 3 loop unrolls in the ExtractMin experiment. Therefore, for this experiment, we are considering only

			5	7	10	12	15	17	20
LList	contains	JF	00:01	02:00	TO	TO	TO	TO	TO
		T	00:03	00:04	00:05	00:06	00:07	00:09	00:15
	insert	JF	00:02	04:56	TO	TO	TO	TO	TO
		T	00:04	00:05	00:07	00:08	00:13	00:26	00:40
	remove	JF	00:04	21:51	TO	TO	TO	TO	TO
		T	00:04	00:06	00:11	00:12	00:17	00:33	00:42
AList	contains	JF	00:02	05:01	TO	TO	TO	TO	TO
		T	00:04	00:06	00:16	00:22	00:27	00:58	02:49
	insert	JF	00:03	11:52	TO	TO	TO	TO	TO
		T	00:04	00:05	00:07	00:08	00:12	00:16	00:25
	remove	JF	00:18	73:27	TO	TO	TO	TO	TO
		T	00:05	00:06	00:17	00:31	01:08	03:13	08:24
CList	contains	JF	00:05	10:23	TO	TO	TO	TO	TO
		T	00:11	00:19	01:23	01:56	05:51	07:25	06:54
	insert	JF	00:20	201:54	TO	TO	TO	TO	TO
		T	00:09	00:12	00:16	00:28	01:07	02:01	04:57
	remove	JF	02:28	TO	TO	TO	TO	TO	TO
		T	00:27	00:59	03:26	03:43	28:18	57:23	89:17
BSTree	contains	JF	09:41	TO	TO	TO	TO	TO	TO
		T	00:01	00:25	114:06	TO	TO	TO	TO
	insert	JF	03:46	TO	TO	TO	TO	TO	TO
		T	00:01	00:26	32:58	TO	TO	TO	TO
	remove	JF	OofM	OofM	OofM	OofM	OofM	OofM	OofM
		T	08:19	102:46	TO	TO	TO	TO	TO
TreeSet	find	JF	00:42	117:49	TO	TO	TO	TO	TO
		T	00:04	00:10	01:56	12:43	58:54	305:06	TO
	insert	JF	OofM	OofM	OofM	OofM	OofM	OofM	OofM
		T	00:43	08:44	TO	TO	TO	TO	TO
AVL	find	JF	00:26	190:10	TO	TO	TO	TO	TO
		T	00:03	00:06	00:36	01:41	08:20	33:06	179:53
	findMax	JF	00:06	49:49	TO	TO	TO	TO	TO
		T	00:01	00:01	00:03	00:04	00:09	00:13	01:09
	insert	JF	OofM	OofM	OofM	OofM	OofM	OofM	OofM
		T	00:07	00:34	04:47	21:53	173:57	TO	TO
BHeap	findMin	JF	00:22	83:07	TO	TO	TO	TO	TO
		T	00:05	00:08	00:14	00:17	01:31	02:51	07:26
	decKey	JF	01:48	TO	TO	TO	TO	TO	TO
		T	00:16	01:13	30:26	TO	TO	TO	TO
	insert	JF	73:47	TO	TO	TO	TO	TO	TO
		T	01:54	08:08	37:30	218:13	TO	TO	TO

Table 8.4. Comparison of code analysis times for 10 loop unrolls using JForge (JF) and TACO (T).

3 loop unrolls for JForge, TACO$^-$ and TACO.

In order to compare with tools based on model checking and SMT-solving, we will carry out the following experiments. We will choose the most complex mutants for each method. For class `LList` we chose mutant AOIU_1, the only mutant of method `Remove` that cannot be killed using scope 2 (it requires scope

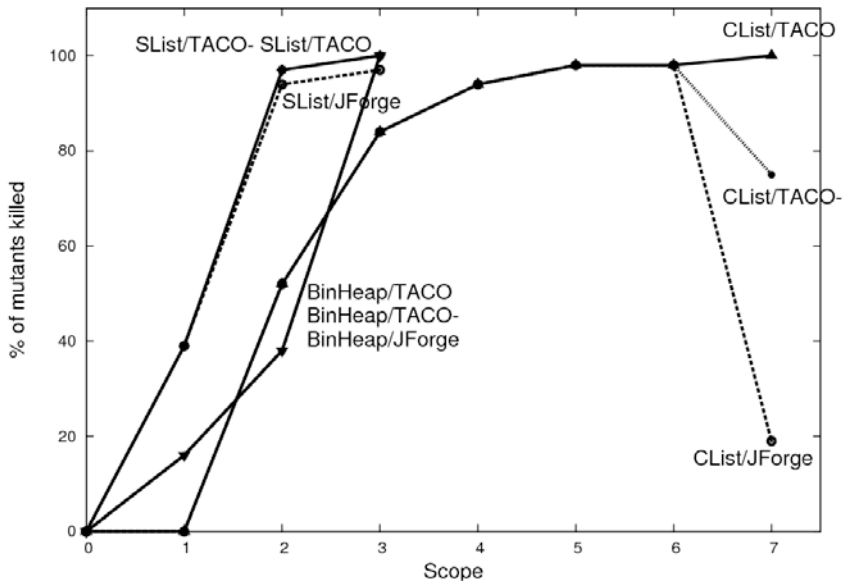


Figure 8.8. Efficacy of JForge, TACO$^-$ and TACO for mutants killing.

	JForge	TACO$^-$	TACO
LList.Remove	01:49	06:56	08:36 + 00:40
CList.Remove	891:50	245:12	34:51 + 06:35
BHeap.ExtractMin	04:34	19:35	16:06 + 01:09

Table 8.5. Analysis times for mutants killing. TACO times reflect the analysis time plus the bounds computation time.

3). For class `CList` we chose mutants AOIS_31 and AOIS_37, the only ones that require scope 7 to be killed. Finally, for class `BHeap` there are 31 mutants that require scope 3 to be killed (all the others can be killed in scope 2). These can be grouped into 7 classes, according to the mutation operator that was applied. We chose one member from each class. In Table 8.6 we present analysis times using all the tools. Table 8.6 shows that TACO, Java PathFinder and Kiasan were the only tools that succeeded in killing all the mutants. Since the fragment of JML supperted by ESC/Java2 is not expressive enough to model the invariant from class BHeap, we did not run that experiment.

	JForge	TACO$^-$	ESCJ	Kiasan	JPF	TACO
LList.AOIU_1	00:01	00:09	00:06	00:05	00:02	00:18
CList.AOIS_31	TO	TO	TO	00:13	02:55	01:00
CList.AOIS_37	TO	TO	TO	00:14	02:18	01:02
BHeap.AOIS_41	00:08	00:13	–	00:32	00:03	00:13
BHeap.AOIU_8	00:02	00:14	–	00:26	00:04	00:12
BHeap.AORB_10	00:04	00:14	–	00:26	00:24	00:12
BHeap.COI_22	00:01	00:11	–	01:05	00:03	00:10
BHeap.COR_5	00:01	00:08	–	00:15	00:25	00:10
BHeap.LOI_15	00:02	00:11	–	00:26	00:29	00:15
BHeap.ROR_23	00:01	00:11	–	00:16	00:04	00:09

Table 8.6. Comparison of analysis behavior for some selected mutants. Analysis time for TACO includes the time required to compute the tight bound amortized among the mutants in each class.

Detecting a Seeded Non-Trivial Bug

Notice that in the previous section, although we choose the supposedly most complex mutants, these are still simple in the sense that they can be killed using small scopes. In this section we are interested in studying the performance of these tools in a context where a greater amount of nodes are needed to find a violation of the specification. In this sense, we focus on the linked data structure for class `CList`. This data structure is composed by the actual (circular) list, and a singly linked list (the cache). The cache list has a maximum size "`maximumCacheSize`" (`maxCS`), set in the actual code to a default value of 20 nodes. When a node is removed from the circular list, it is added to the cache (unless the cache is full). Let us consider the code snippet from `remove` presented in Fig. 8.9.(a). Figure 8.9.(b) gives us a bug-seeded version. A failure occurs the bug-seeded code when a node is removed and the cache is full. In effect, if the maximum cache size is set to the default of 20, a $21^{st.}$ element can be added to the cache. This leads to a violation of the invariant that constrains the cache size to be at most the value of the maximum cache size field.

In Table 8.7 we report analysis information after looking for the bug in the bug-seeded code (BS), for varying numbers of loop unrolls in method `super.removeNode`. We have tailored the bug-seeded code (and its contract), to be analyzed using the same tool set we have applied in the previous section for analysing the more

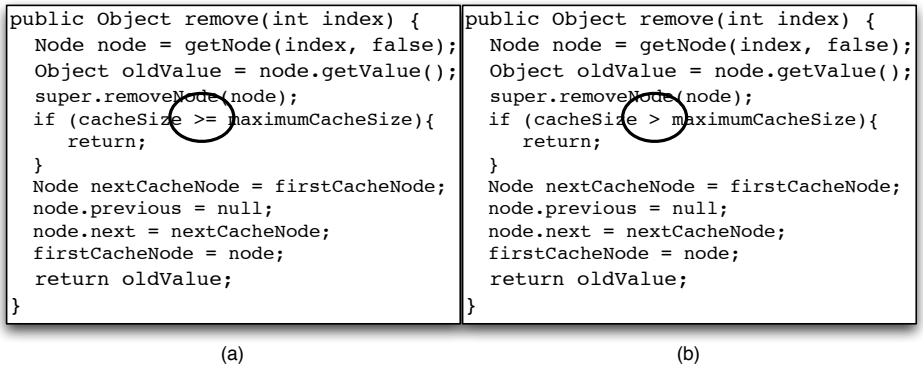


Figure 8.9. Code snippets from `CList.remove` (a), and a bug-seeded version (b).

complex mutants.

We computed a bound for TACO in 27:04 using one iteration of the iterative algorithm of Fig. 7.3. Table 8.7 shows that many times it is not necessary to compute the tightest bound, but rather thin the initial bound with a few iterations of the algorithm in order to achieve a significant speed up in analysis time. The debugging process consists on running a tool (such as TACO, JForge, etc.) and, if a bug is found, correct the error and start over to look for further bugs. Unlike JForge (where each analysis is independent of the previous ones), the same bound can be used by TACO for looking for all the bugs in the code. Therefore, the time required for computing the bound can be amortized among these bugs. Since the bound does not depend on the number of unrolls, in Table 8.7 we have divided 27:04 among the 7 experiments, adding 03:52 to each experiment. Time is reported as "bound computation time" + "SAT-solving time".

We also compared with Boogie [6] using Z3 [65] as the back-end SMT solver. In order to produce Boogie code we used Dafny [60] as the high-level programming and specification language. When ran on the bug-seeded code with 10 loop unrolls, Boogie produced in the order of 50 warning messages signaling potential bugs. A careful inspection allowed us to conclude that all warnings produced by Boogie were false warnings.

Since most tools failed to find the bug with `maxCS` = 20, we also considered a version of the code with up to 2 loop unrolls and varying values for `maxCS`; in this way the bug can be found in smaller heaps. Table 8.8 reports the corresponding analysis times. In TACO we have restricted the algorithm that computes the

LU	JForge	ESC/Java2	JPF	Kiasan	TACO
4	OofM(227)	OofM(206)	TO	OofM(4)	03:52 + 03:56
6	TO	OofM(207)	TO	OofM(4)	03:52 + 31:14
8	OofM(287)	OofM(213)	TO	OofM(4)	03:52 + 33:23
10	05:40:22	OofM(215)	TO	OofM(4)	03:52 + 00:11
12	06:53:04	OofM(219)	TO	OofM(4)	03:52 + 03:30
15	24:08	OofM(219)	TO	OofM(4)	03:52 + 15:00
20	TO	OofM(218)	TO	OofM(4)	03:52 + 00:06

Table 8.7. Outcome of the analysis `maxCS` = 20. Ten hours timeout. TACO's bound computation is amortized.

bound for each scope to run at most 30 minutes.

mCS	JForge	ESC/Java2	JPF	Kiasan	TACO
5	00:13	OofM(187)	00:07	00:18	01:21 + 00:01
10	05:13	OofM(212)	00:20	00:43	02:25 + 00:11
13	OofM(529)	OofM(221)	00:38	OofM(3)	05:27 + 00:32
15	OofM(334)	OofM(214)	00:53	OofM(3)	21:31 + 00:15
18	14:04	OofM(200)	01:27	OofM(4)	30:00 + 02:27
20	OofM(494)	OofM(556)	02:17	OofM(4)	30:00 + 02:11

Table 8.8. Up to 2 unrolls and varying `maxCS`. 10 hours timeout.

The code has a fault that requires building a non-trivial heap to expose it. The technique introduced in this article made TACO the *only* tool capable of finding the bug in *all* cases reported in Tables 8.7 and 8.8. When the size of the code is small (2 loop unrolls in Table 8.8), tools based on model checking were able to find the bug. They failed on larger code, which shows that in the example TACO scales better. Tools based on SMT solving systematically failed to expose the seeded bug.

Detecting a Previously Unknown Fault

As we mentioned in Galeotti et al. [40], TACO found a previously unreported bug in method `ExtractMin` of class `BHeap`.

A distinguishing characteristic of this fault is that it cannot be reproduced using mutation because the smallest input that produces a failure has 13 nodes,

and as we showed before in Sec. 8.6, all mutants were killed with only 3 nodes. Another interesting attribute of this defect is that it is not easily identified as a bug introduced as a programmer typo. We localize the fault in the helper method `merge` (Listing 8.1). The defect is that the sibling pointing to `temp1` is not updated when *degree*(*temp*1) > *degree*(*temp*2). More specifically, field `sibling` from the previous sibling of the object referenced by `temp1`(at line 24) should point to the object referenced by `temp2` (at line 24).

As the reader may notice, the fault is not trivially discovered by inspecting the code.

Listing 8.1. Buggy implementation of merging two binomial heaps

```
1  void merge(BinomialHeapNode binHeap) {
2
3    BinomialHeapNode temp1 = Nodes, temp2 = binHeap;
4    while ((temp1 != null) && (temp2 != null)) {
5      if (temp1.degree == temp2.degree) {
6        BinomialHeapNode tmp = temp2;
7        temp2 = temp2.sibling;
8        tmp.sibling = temp1.sibling;
9        temp1.sibling = tmp;
10           temp1 = tmp.sibling;
11         } else {
12           if (temp1.degree < temp2.degree) {
13             if ((temp1.sibling == null)
14                 || (temp1.sibling.degree>temp2.degree)) {
15               BinomialHeapNode tmp = temp2;
16               temp2 = temp2.sibling;
17               tmp.sibling = temp1.sibling;
18               temp1.sibling = tmp;
19               temp1 = tmp.sibling;
20             } else {
21               temp1 = temp1.sibling;
22             }
23           } else {
24             BinomialHeapNode tmp = temp1;
25               temp1 = temp2;
26               temp2 = temp2.sibling;
27               temp1.sibling = tmp;
28               if (tmp == Nodes) {
```

```
29                      Nodes = temp1;
30                    }
31                }
32              }
33    }
34
35    if (temp1 == null) {
36      temp1 = Nodes;
37            while (temp1.sibling != null) {
38              temp1 = temp1.sibling;
39            }
40            temp1.sibling = temp2;
41    }
42  }
```

The input datum leading to the failure is presented in Fig. 8.10. Notice that at least 4 loop unrolls and 13 node elements were required in TACO in order to exhibit the failure. The failure is not found if a smaller scope of analysis is used. When attempting to discover the bug using all the tools, JForge, TACO$^-$ and JPF reached the 1 hour time limit. On the other hand, Kiasan exhausted the RAM memory. TACO was the only tool that succeeded in discovering the error. The analysis time for computing the bound was 20 minutes and 13 seconds, plus the analysis time for the method itself was 53 seconds.

8.7 Threats to Validity

We begin by discussing how representative the selected case studies are. As discussed by Visser et al. [86], container classes have become ubiquitous. Therefore, providing confidence about their correctness is an important task in itself. But, as argued by Siddiqui et al. [78], these structures (which combine list-like and tree-like structures) are representatives of a wider class of structures including, for instance, XML documents, parse trees, etc. Moreover, these structures have become accepted benchmarks for comparison of analysis tools in the program analysis community (see for instance [11, 27, 52, 76, 86]).

In all experiments we are considering the performance of TACO$^-$ as a control variable that allows us to guarantee that TACO's performance improvement is due to the presented techniques.

In Section 8.5 we analyzed bug-free code. Since the process of bug finding ends when no more bugs are found, this situation where bug free code is analyzed

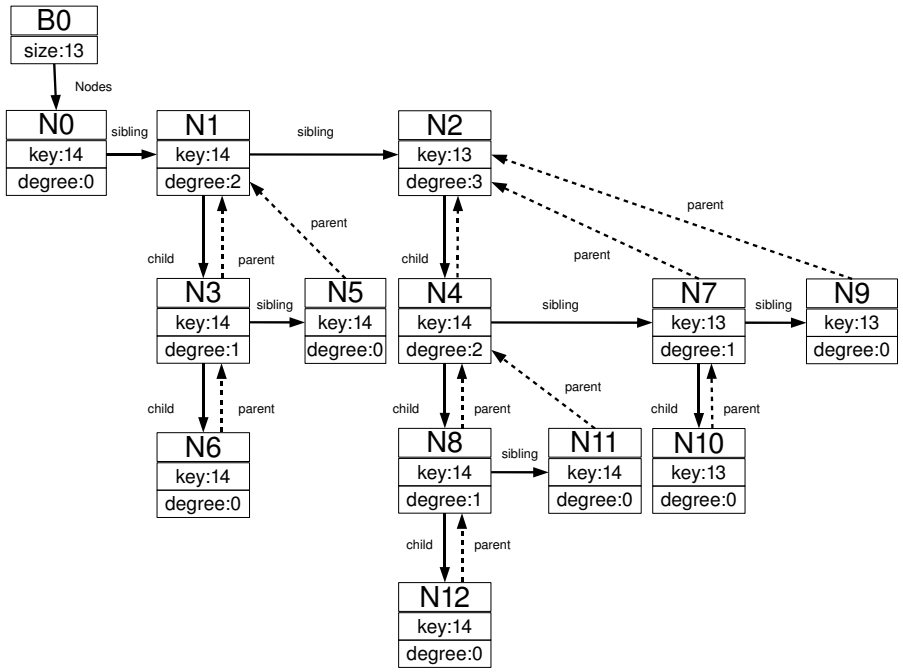


Figure 8.10. A 13 nodes heap that exhibits the failure in method `ExtractMin`**.**

is not artificial, but is rather a stress test that necessarily arises during actual bug finding.

In Section 8.6 we have compared several tools. It is not realistic to claim that every tool has been used to the best of its possibilities. Yet we have made our best efforts in this direction. In the case of JForge, since it is very close to TACO, we are certain we have made a fair comparison. For Java PathFinder and Kiasan we were careful to write repOK invariant methods in a way that would return false as soon as an invariant violation could be detected. For ESC/Java2, since it does not support any constructs to express reachability, we used weaker specifications that would still allow the identification of bugs. For Jahob we used Jahob's integrated proof language, and received assistance from Karen Zee in order to write the models. More tools could have been compared in this section. Miniatur and FSoft are not available for download even for academic use, and therefore were not used in the comparison. Other tools such as CBMC and Saturn (designed for analysis of C code) departed too much from our intention to compare tools for the analysis of Java code.

Analysis using TACO requires using a cluster of computers to compute tight bounds. Is it fair to compare with tools that run on a single computer? While we do not have a conclusive answer, for the bug in method `ExtractMin` (even considering the time required to compute the bounds *sequentially*) TACO seems to outperform the sequential tools. This is especially clear in those cases where the sequential tools run out of memory before finding the bug (as is the case for Kiasan and JForge). More experiments are required in order to provide a conclusive answer.

8.8 Chapter summary

This chapter presented several experiments. They were conducted with two aims: assessing the efficiency of the proposed technique, and comparing TACO against other similar tools based on SMT-Solving and model checking.

First, the experiments allow us to conclude that the sole inclusion of the symmetry breaking predicates speeds up the analysis considerably. This result is especially interesting if no cluster facilities are available for computing tight bounds.

Second, we presented an evaluation of two parallel algorithms for computing tight bounds. From studying these results, two conclusions arise:

- The computation of the tightest bound is affordable for the scopes under analysis using both algorithms.

- The eager algorithm exhibits a significant improvement in performance compared to the iterative algorithm, although no improvement is achieved in asymptotic terms.

Third, we conclude that the SAT-Solving analysis is sensitive to tightening the upper bounds.

Finally, TACO outperforms JForge and TACO$^-$ when analyzing bug-free as well as faulty software (i.e., mutant-generated and manually seeded programs). TACO also outperforms in these experiments Java PathFinder, Kiasan, Dafny and ESC/Java2.

Several threats to validity were discussed to generalize the validity of the results.

Chapter 9

Related work

In this chapter we present a brief introduction to the state-of-the-art at SAT-based program verification and (to the best of our knowledge) the most related work.

9.1 Java SAT-based bounded verification

JAlloy

JAlloy [84] is a SAT-based program analysis tool for Java programs. JAlloy performs whole program analysis, creating an intermediate Alloy representation. In this intermediate representation, new relations and variables are declared to store the values of modified fields and Java variables. The control flow graph is explicitly modelled using boolean variables. Specification in JAlloy are written in a stylized version of Alloy. The current JAlloy implementation includes some optimizations for modelling storing and accessing Java fields.

The tool was tested on red-black trees and an implementation of the Shorr-Waite garbage collection algorithm. At the current moment, no JAlloy prototype is publicly available for download.

Karun

Karun [80] is motivated by the observation that inlining does not scale for large programs. Loops are desugared into recursive procedures and every call-site is replaced with a procedure summary (or abstraction). Following the *counter-example guided abstraction refinement* (CEGAR) paradigm [44], program summaries are refined using the UNSAT core provided by the SAT-solver. Karun is intended to verify partial (or weak) properties. As explained in Dennis PhD

dissertation [26], the complexity of the property under analysis may lead to the full inline of the behavior of the invoked procedure. Under these circumstances, the analysis would have performed better by inlining the called methods. Similar to JAlloy, Karun's specification language is a fragment of the Alloy language.

Karun was used with two industrial size case studies: the QUARTZ API and the OpenJGraph API. As with JAlloy, no Karun prototype is available for download.

Forge

Instead of creating an Alloy intermediate representation, Forge [26] works by translating the program under analysis (and the property to be verify) to the current Alloy's backend, Kodkod [81]. This decision was taken mainly because of the ability to describe partial instances provided by the Kodkod API. One place where they are exploited is in fixing a total ordering over each domain to optimize analysis by eliminating symmetries when analyzing programs using dynamic allocation. Forge's translation of the procedure to logic uses a symbolic execution technique [55], which means that no control flow graph is explicitly modelled. Loops are desugared by unwinding to a user provided bound.

Two front-ends are currently available for using Forge: JMLForge and JForge. JMLForge allows one to write specifications in the JML language. JForge's specification language is a combination of JML and the Alloy Annotation Language (AAL) [56], known as JForge Specification Language (JFSL) [91]. Forge performs modular analysis, which means that the specification of the called methods is used instead of the actual implementation while analyzing a program. A distinguishing feature of Forge is that, in case no counterexample is found within the provided scope of analysis, the UNSAT core is used to present the user with a highlighting of the program and the specification under analysis. This *coverage metric* attempts to identify statements that were "missed" (not covered) by the bounded verification.

Forge was used to verify the correctness of several linked-list implementations [27] and the KOA electronic voting system [28]. Both front-ends for Forge (JForge and JMLForge) are currently available for download [37].

Miniatur

Miniatur [31] is another SAT-based whole program analysis tool. The specification language is a relational language similar to Alloy. Miniatur encoding exploits the static single assignment (SSA) [23] form and control dependence

graph (CDG) [23] to efficiently encode data- and control-dependence information. Due to this, Miniatur is able to perform a slicing at the logical formula level with respect to the specification to be checked. Notice that, since slicing plays a key role in Miniatur's analysis, the intended properties to be checked using Miniatur are partial properties (as with Karun).

Miniatur also introduces a logarithmic encoding for integers and a efficient support for sparse arrays. This allows Miniatur to handle realistic fragments of code that manipulate both heap-allocated objects and integers. Both optimizations are not incompatible with our translation and they are intended to be include in a future version of TACO. Finally, as with JAlloy and Karun, no Miniatur prototype is available for public use.

9.2 C SAT-based bounded verification

CBMC

CBMC [20] is a Bounded Model Checker for ANSI-C and C++ programs. It allows verifying user-specified C language assertions and other assertions such as array bounds (buffer overflows), pointer safety and exceptions. Since CBMC targets user-specified assertions, the properties a user may define could be as rich as he or she may want. A limitation of CBMC is that it only analyzes closed systems, which means the method under analysis must have no arguments. What is more, any access to a reference beyond the heap space allocated by the program under analysis is considered as a possibly faulty access. In order to finitize the code, the user should provide a bound to unwind loops and recursive calls. Notice that, since it is assume a closed-system analysis, no scope to bound heap objects is required. The tool is publicly available at [19]

SATABS

SATABS [21] was developed as a enhancement of the CBMC tool. It performs predicate abstraction on the original code and passes the abstract model to a model checker. If an abstract counterexample is found, a SAT-solver is used to check its feasibility. If the SAT instance is unsatisfiable, the produced resolution proof is used to refine the predicates exploiting that, for propositional logic, a propositional Craig Interpolant can be extracted from a resolution proof in linear time.

SATURN

SATURN [90] builds a single SAT formula that is fed to the SAT-Solver as CBMC does. It uses as its main techniques a slicing algorithm and function summaries. Assertions must be written in C language. At the intraprocedural level, sequential code is faithfully modeled (no abstractions are used). On the contrary, at the interprocedural level it performs slicing by replacing function calls with automatically generated procedure summaries.

The scalability and precision of SATURN depends on *summarizing* the effect of a function call through the function summary. The user must provide a description of how summaries are built. In order to do so, a summary design language (named CALIPSO) is given. The description of the right summary function represents a challenge even for expert users, specially for complex properties of linked data structures.

As SATURN stores function summaries in a repository for future use, it is similar to TACO in storing information to amortize its cost during future analysis. SATURN's cluster architecture also allows to distribute the analysis for single functions into several workers (rerunning analysis if dependencies are found).

F-Soft

F-Soft [46] also analyzes closed-systems written in C language. The user may provide a specification by annotating a given statement with a distinguishing C label. F-Soft integrates SAT-based verification with several static analysis techniques and predicate abstraction to alleviate the SAT-solving process. User should provide a bound to the number of unrolls and to the number of dynamically allocated objects. F-Soft works with a customized SAT-solver that allows (as program counter is explicitly modelled) to increase the likelihood that the SAT-Solver first decides values for those propositional variables modelling the program counter.

Similarly to TACO, F-Soft computes lower and upper bounds for values of integer valued variables and for pointers under the hypothesis that runs have bounded length. It is based on the framework presented in Rugina et al. [73]. Our technique produces tighter upper bounds because it does not compute feasible intervals for variables, but instead checks each individual value. Regrettably, a comparison with this tool was not possible due to its unavailability.

9.3 Theorem Proving

SMT-based program verification

SMT-Solvers are extensions of SAT-Solvers that handle different logical theories such as equality with uninterpreted functions, arithmetic and arrays. Several software verification tools (such as ESCJava2 [13], Spec# [8] and HAVOC [14]) rely on a SMT-Solver as its back-end decision procedure.

ESC/Java2 [13] works by translating a JML specification to a verification condition that is later fed to the Simplify SMT-Solver [29]. It supports a rather large fragment of JML language for specifying a method's behaviour. This allows users to write weak specifications as well as strong specifications. If the user decides to avoid writing loop invariant annotations, a loop unwinding to a certain user-provided bound is performed. As Forge, ESC/Java2 performs a modular analysis. Since it relies on a SMT-Solver, no user-provided scope is needed besides the loop bound. In order to make analysis more automated, ESC/Java2 renounce to be neither complete nor sound. This means that, due to its translation to FOL, false warnings may be reported, and also real bugs may be missed. As SAT-based tools faithfully represent the bounded space of instances they search, their output tend to be void of false warnings.

Spec# [8] allows to verify specifications written for C# programs. Although Spec#'s specification language is less expressive than JML, this follows an explicit design goal: maintain the specification language as expressive as the verification back-end allows. Spec# creates a single BoogiePL specification [6] that is later fed to the Z3 SMT-Solver [65]. It also adopts a series of language restrictions in order to enforce the modular verification of invariants. By following this programming discipline, only the invariant corresponding to the class under analysis is verified, and the other invariants are assumed to be preserved. This provides a framework in which problems such as reentrancy and nested structures are solved. Spec# also successfully handles recursive invariants, mutual recursions and ownership transfer.

HAVOC [14] is a modular checker for ANSI-C programs. Similarly to Spec#, HAVOC transforms the C program as well as the assertion to be verified into a BoogiePL which is later fed to the Z3 SMT-Solver. HAVOC specially targets the verification of functional properties of linked data structures such as lists and arrays. In order to allow the specification of such properties, a reachability predicate is provided. HAVOC also provides a faithful operational semantics for C programs accounting for the low-level operations in systems code (i.e., pointer arithmetic, bitwise operations, etc.). A distinguishing feature of HAVOC is its

contract inference mechanism to reduce the user burden writing contracts [59].

Dafny [60] had started as an experiment to encode dynamic frames [53], but it has grown to become more of a general-purpose specification language and verifier (where modular verification is achieved via dynamic frames). By adding the right loop invariants, Dafny is able to verify the correctness of the Schorr-Waite garbage collecting algorithm in less than 5 seconds. Nevertheless, it depends strongly on writing an appropriate set of SMT-amendable loop invariants.

Jahob

Instead of relying on a single monolithic prover, Jahob [10] splits the verification condition into several verification goals which are sequentially fed to several first-order provers (such as SPASS [88] and E [75]) and SMT-Solvers (CVC3 [41] and Z3 [65]). Jahob allows the full functional verification of complex properties over linked data structures such as binary trees, red black trees, etc.

As Jahob's language was designed as a proof language, it provides language constructs for identifying lemmas, witnesses of existential quantifications, patterns for instantiating universal quantifiers, proofs by induction, etc. Although the expressiveness of this proof language allows the user to write very useful annotations for the underlying decision procedures (which allows verifying very complex properties), it is easy to see that annotation goes far beyond specifying program's behaviour.

In our experience, by solely providing a the a program's behaviour specification Jahob neither succeeded in verifying the provided specifications, nor provided an understandable counterexample (warnings were expressed in terms of the back-end decision procedures).

9.4 Model checking Java programs

Java PathFinder [85], is a whole program analysis explicit model checker. Java PathFinder mimics the Java Virtual Machine interpreting the compiled Java bytecode. In order to avoid state-space explosion, the tool uses partial order and symmetry reduction, slicing and abstraction. Java PathFinder checks for deadlocks, safety properties, user-defined assertions written in Java, as well as Linear Time Temporal Logic (LTL) properties.

Due to its explicit (or concrete) nature, Java PathFinder is restricted to closed-systems only (method under analysis must have no arguments). If the user desires to verify an open-system (such as a library or an API), he or she should pro-

vide a verification harness using the non-deterministic infrastructure provided by Java PathFinder to build non-deterministically an input state.

JPF-SE [1] is an extension to the core Java PathFinder implementation. In this extension, concurrent Java programs treats unbound inputs as "symbolic" values, allowing the verification of open-systems with no need of a verification-harness. Unknown heap data structures are instantiated on demand using lazy initialization [58]. JPF-SE works by symbolically executing the Java bytecode, whenever a path condition is updated, it is checked for satisfiability using several decision procedures (YICES [32], CVC-Lite [22], Omega [72], STP [12]). If the path condition is not satisfiable, the model checker backtracks. In order to assure the termination of the symbolic model checking, a bound to search depth, input size and length of symbolic formulas should be provided.

In Anand et al. [2] the authors of the JPF-SE extension consider abstraction techniques for computing and storing abstract states in order to help the termination of the symbolic model checking.

Kiasan [9] was originally inspired in JPF-SE. As JPF-SE, Kiasan is able to check complex properties, more specifically, complex linked structures properties written in JML. Kiasan is built on top of the Bogor model-checking framework [71]. As JPF-SE, Kiasan is essentially a symbolic Java Virtual Machine that invokes the CVC-Lite SMT-Solver for checking the feasibility of the current path condition.

To ensure its termination, a user is not only able to bound the number of loop iterations and the length of the call chains, but also to bound the length of any chain of object references and the number of unique array indices the program may use. As already observed in Greg Dennis PhD dissertation [26], we also believe this represents a more gentle approach to termination that bounding the size of a path condition formula. Kiasan also introduces a customized version of the lazy initialization presented in Khurshid et. al. [58]. Finally, for each path traversed, Kiasan generates the corresponding test input in JUnit format.

9.5 Related heap canonization

The idea of canonicalizing the heap in order to reduce symmetries is not new. In the context of explicit state model checking, the articles [45, 68] present different ways of canonicalizing the heap ([45] uses a depth-first search traversal, while [68] uses a breadth-first search traversal of the heap). The canonicalizations require modifying the state exploration algorithms, and involve computing hash functions in order to determine the new location for heap objects in the canonicalized heap. Notice that:

- The canonicalizations are given algorithmically (which is not feasible in a SAT-solving context).
- Computing a hash function requires operating on integer values, which is appropriate in an algorithmic computation of the hash values, but is not amenable for a SAT-solver.

In the context of SAT-based analysis, [57] proposes to canonicalize the heap, but the canonicalizations have to be provided by the user as ad-hoc predicates depending on the invariants satisfied by the heap. JForge [26] exploits the fact that allocation follows a total ordering of the atoms in the domain (by convention the lexical ordering). Nevertheless, this optimization is restricted to which values a variable may refer to when allocating fresh heap memory objects.

9.6 Related extensions to Alloy

As our dynamic extension to Alloy, Imperative Alloy [70] was defined with the specific purpose of providing Alloy users with a unified framework for specifying and analyzing dynamic systems. Similarly to DynAlloy, Imperative Alloy works by translating this extension to a plain Alloy model which is later analyzed using the Alloy engine.

Nevertheless, DynAlloy is based in dynamic logic [43] instead of relational constructs. As DynAlloy, Imperative Alloy allows the user to specify atomic actions, and more complex actions such as loops, sequential composition, etc. Imperative Alloy's translation into plain Alloy does explicitly model the time instant using atoms. On the other hand, DynAlloy's translation uses several variables to model each value update. Therefore, Imperative Alloy provides temporal constructs for specifying safety and liveness properties.

9.7 Related tight bound computation

F-Soft's range analysis intends to reduce the number of propositional variables in the resulting SAT formula. In this sense, F-Soft is related to our work on distributed bound computation. Likewise, MemSAT [82] computes a set of lower and upper bounds on the space to be searched for relational variables. As TACO, MemSAT's back-end is the Kodkod model finder [81]. MemSAT is a diagnosis tool explicitly created to find inconsistencies in obeying a certain memory model. As TACO, MemSAT needs to bound the search space (i.e., number of loop unrolls).

9.8 Shape Analysis

Unlike techniques based on abstraction that require the user to provide *core properties* (i.e., *shape analysis* [74]), TACO does not require user-provided properties other than the JML annotations.

Chapter 10

Conclusions

In this chapter we discuss the conclusions and future work of the dissertation.

10.1 Conclusions

The contributions of this dissertation are twofold. First, we have defined an extension to Alloy that allows the user to reason about traces. We believe that using actions within Alloy favours a better separation of concerns, since models do not need to be reworked in order to describe the adequate notion of trace modelling the desired behaviour. Using actions, the problem reduces to describing how actions are to be composed.

We have provided an intermediate representation between DynAlloy and JML. We have also shown how to appropriately translate a JML annotated Java program into its DynJML counterpart. DynJML allows us to have a description closer to DynAlloy. We have presented a tool TACO which implements all these translation phases from JML to SAT.

Second, this dissertation also presents a novel methodology based on:

1. adding appropriate constraints to SAT problems, and
2. using the constraints to remove unnecessary variables.

This new methodology makes SAT-solving a method for program analysis as effective as model checking or SMT-solving when analyzing strong specifications of linked data structures.

10.2 Future work

The experimental results presented in the thesis show that bounds can be computed effectively, and that once bounds have been computed, the analysis time improves considerably. This allowed us to analyze real code using domain scopes beyond the capabilities of current similar techniques and find bugs that cannot be detected using state-of-the-art tools for bug-finding. Still, while this dissertation presents a new approach to bound computation with respect to that presented in Galeotti et al. [40], we are working on a new, and more efficient, method for distributed bound computation.

Although it was mentioned in this dissertation, a more extensive comparative study between Imperative Alloy and DynAlloy is required. We are specially interested in assessing scalability and efficiency of both extensions to Alloy.

As parallel computing facilities become commonplace, possible applications of SAT-based software verification arise. We are developing a tool for parallel analysis of Alloy models called ParAlloy. Preliminary results show that combining TACO with ParAlloy will produce a new speed-up of several orders of magnitude with respect to the times presented in this dissertation. It is worth emphasizing that the kind of infrastructure we are using during parallel computation of bounds is inexpensive, and should be accessible to many small or medium sized software companies.

The optimization technique introduced in this dissertation limits itself to bound those propositional variables which represent the *initial* state of the analysis. An interesting development will require to perform a data flow analysis in order to extended the bound to those variables which represent *intermediate* computation stages. The intuition is that, when field updates occur, the SAT-solver is not able to recognize the order in which the program control flows. We believe that a conservative bounding to intermediate states will have a tremendous impact on increasing the scalability of the technique.

The symmetry breaking and bound computation techniques presented in the thesis are quite general. Explicit state model checkers (such as JPF) can use tight bounds in order to prune the state space when a state contains edges that lay outside the bound. Korat [11] can avoid evaluating the `repOk` method whenever the state is not contained in the bounds. Running a simple membership test will many times be less expensive than running a `repOk` method. Tools that are similar to TACO (such as Miniatur and JForge), can make direct use of these techniques.

Finally, the technique suits perfectly into SAT-based test input generation as well. An empirical assessment of its capabilities comparing to other state-of-the-

art test tools will provide more insight on the exact dimension of the contributions of this dissertation.

Bibliography

[1] Anand S., Pasareanu C. S., Visser W. *JPF-SE: A Symbolic Execution Extension to Java PathFinder*. In proceedings of TACAS 2007. pp. 134–138.

[2] Anand S., Pasareanu C. S., Visser W. *Symbolic execution with abstraction*. In proceedings of STTT 2009. pp. 53–67.

[3] Aho A., Sethi R., Ullman J. *Compilers: Principles, Techniques and Tools* Addison Wesley. 1986

[4] Appel A. *Modern compiler implementation in Java* Cambridge University Press. 1998.

[5] Back R. *A calculus of refinements for program derivations*. Acta Informatica, 25(6) pages 593–624, August 1988.

[6] Barnett M., Chang B.E, DeLine R., Jacobs B., Leino K.R.M. *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. FMCO 2005: pp. 364–387.

[7] Barnett, M., Fändrich, M., Garbervetsky, D., Logozzo, F.: *Annotations for (more) precise points-to analysis*. IWACO 2007: ECOOP International Workshop on Aliasing, Confinement and Ownership in object-oriented programming. July 2007.

[8] Barnett M., Leino K. R. M., Schulte W., *The Spec# programming system: An overview*. In CASSIS 2004, LNCS vol. 3362, Springer, 2004.

[9] Belt, J., Robby and Deng X., *Sireum/Topi LDP: A Lightweight Semi-Decision Procedure for Optimizing Symbolic Execution-based Analyses*, FSE 2009, pp. 355–364.

[10] Bouillaguet Ch., Kuncak V., Wies T., Zee K., Rinard M.C., *Using First-Order Theorem Provers in the Jahob Data Structure Verification System*. VMCAI 2007, pp. 74–88.

[11] Boyapati C., Khurshid S., Marinov D., *Korat: automated testing based on Java predicates*, in ISSTA 2002, pp. 123–133.

[12] Cadar C., Ganesh V., Pawlowski P. M., Dill D. L., Engler D. R. *Exe: Automatically generating inputs of death*. In Computer and Communications Security, 2006.

[13] Chalin P., Kiniry J.R., Leavens G.T., Poll E. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*. FMCO 2005: 342-363.

[14] Chatterjee S., Lahiri S., Qadeer S., and Rakamaric Z., *A Reachability Predicate for Analyzing Low-Level Software*. In Tools and Algorithms for the Construction and Analysis of Systems (TACAS '07), Springer Verlag, April 2007.

[15] Clarke E., Kroening D., Lerda F., *A Tool for Checking ANSI-C Programs*, in TACAS 2004, LNCS 2988, pp. 168–176.

[16] Cok D. *Reasoning with specifications containing method calls and model fields*, in Journal of Object Technology, vol. 4, no. 8, Special Issue: ECOOP 2004 Workshop FTfJP, October 2005, pp. 77-103

[17] Coldewey, D. *Zune bug explained in detail* `http://www.crunchgear.com/2008/12/31/zune-bug-explained-in-detail/`

[18] Cook, S., *The Complexity of Theorem-Proving Procedures*, in Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 151–158, ACM, 1971.

[19] The CBMC homepage `http://www.cs.cmu.edu/~modelcheck/cbmc/`

[20] Clarke, E., Kroening, D., Lerda F. *A Tool for Checking ANSI-C Programs*. In proceedings of TACAS 2004. LNCS (2988) pp. 168–176.

[21] Clarke, E., Kroening, D., Sharygina, N., Yorav, K. *SATABS: SAT-based Predicate Abstraction for ANSI-C* . In proceedings of TACAS 2005. LNCS (3440) pp. 570–574.

[22] CVCL. `http://www.cs.nyu.edu/acsys/cvcl/`

[23] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. *Efficiently computing static single assignment form and the control dependence graph*. ACM TOPLAS 13, 4 (1991), pp. 451–490.

[24] deMillo R. A., Lipton R. J., Sayward F. G., *Hints on Test Data Selection: Help for the Practicing Programmer*, in IEEE Computer pp. 34–41, April 1978.

[25] Deng, X., Robby, Hatcliff, J., *Towards A Case-Optimal Symbolic Execution Algorithm for Analyzing Strong Properties of Object-Oriented Programs*, in SEFM 2007, pp. 273-282.

[26] Dennis G. *A Relational Framework for Bounded Program Verification*. MIT PhD Thesis. July 2009.

[27] Dennis, G., Chang, F., Jackson, D., *Modular Verification of Code with SAT*. in ISSTA'06, pp. 109–120, 2006.

[28] Dennis, G., Yessenov, K., Jackson D., *Bounded Verification of Voting Software*. in VSTTE 2008. Toronto, Canada, October 2008.

[29] Detlefs D., Nelson G., Saxe J. B. *Simplify: a theorem prover for program checking*. In Journal of the ACM (JACM). Volume 52 Issue 3. May 2005.

[30] Dijkstra E. W., Scholten C. S. *Predicate calculus and program semantics*. Springer-Verlag, 1990

[31] Dolby J., Vaziri M., Tip F., *Finding Bugs Efficiently with a SAT Solver*, in ESEC/FSE'07, pp. 195–204, ACM Press, 2007.

[32] Dutertre B., de Moura L. *A Fast Linear-Arithmetic Solver for DPLL(T)*. In Proceedings of CAV'06, volume 4144 of LNCS, pages 81–94. Springer-Verlag, 2006.

[33] Een, N., Sorensson, N., An extensible SAT-solver. Lecture notes in computer science. Volume 2919 (2004) pages 502–518.

[34] Flanagan, C., Leino, R., Lillibridge, M., Nelson, G., Saxe, J., Stata, R., *Extended static checking for Java*, In PLDI 2002, pp. 234–245.

[35] Frias, M. F., Galeotti, J. P., Lopez Pombo, C. G., Aguirre, N., *DynAlloy: Upgrading Alloy with Actions*, in ICSE'05, pp. 442–450, 2005.

[36] Frias, M. F., Lopez Pombo, C. G., Galeotti, J. P., Aguirre, N., *Efficient Analysis of DynAlloy Specifications*, in ACM-TOSEM, Vol. 17(1), 2007.

[37] Forge website `http://sdg.csail.mit.edu/forge`

[38] Gage D., McCormick J. *"We did nothing wrong" Why software quality matters*. Available from: `http://www.baselinemag.com/print_article2/0,1217,a=120920,00.asp`

[39] Galeotti, J. P., Frias, M. F., *DynAlloy as a Formal Method for the Analysis of Java Programs*, in Proceedings of IFIP Working Conference on Software Engineering Techniques, Warsaw, 2006, Springer.

[40] Galeotti J., Rosner N., Lopez Pombo, C., Frias M., *Analysis of Invariants for Efficient Bounded Verification*, in ISSTA 2010, Trento, Italy.

[41] Ge Y., Barrett C., and Tinelli C. *Solving quantified verification conditions using satisfiability modulo theories*. In CADE, 2007.

[42] Goldberg E. and Novikov Y. BerkMin: A fast and robust sat-solver. In *Proceedings of the conference on Design, automation and test in Europe*, pages 142–149. IEEE Computer Society.

[43] Harel D., Kozen D., and Tiuryn J. *Dynamic logic. Foundations of Computing*. MIT Press, 2000

[44] Henzinger T. A., Jhala R., Majumdar R., McMillan K. L. *Abstractions from proofs*. In POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages.

[45] Iosif R., *Symmetry Reduction Criteria for Software Model Checking*. SPIN 2002: 22-41

[46] Ivančić, F., Yang, Z., Ganai, M.K., Gupta, A., Shlyakhter, I., Ashar, P., *F-Soft: Software Verification Platform.* In CAV'05, pp. 301–306, 2005.

[47] Jackson D., *A micromodels of software: Lightweight modelling and analysis with Alloy*. MIT Laboratory for Computer Science, Cambridge, MA, 2002.

[48] Jackson D., Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 2002.

[49] Jackson, D., *Software Abstractions.* MIT Press, 2006.

[50] Jackson D., Damon C., *Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector*. ISSTA 1996: 239–249.

[51] Jackson D., Shlyakhter, and Sridharan M., A micromodularity mechanism. In *Proceedings of the 8th European software engineering conference held together with the 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 62–73, Vienna, Austria, 2001. Association for the Computer Machinery, ACM Press.

[52] Jackson, D., Vaziri, M., *Finding bugs with a constraint solver*, in ISSTA'00, pp. 14-25, 2000.

[53] Kassios I. T. *Dynamic frames: Support for framing, dependencies and sharing without restrictions*. In FM 2006: Formal Methods, 14th International Symposium on Formal Methods, volume 4085 of Lecture Notes in Computer Science, pages 268–283. Springer, August 2006.

[54] JMLForge website `http://sdg.csail.mit.edu/forge/jmlforge.html`

[55] King J. C. *Symbolic execution and program testing*. Communications of the ACM, 19(7), pp.385–394, 1976.

[56] Khurshid, S., Marinov, D., Jackson, D., *An analyzable annotation language*. In OOPSLA 2002, pp. 231-245.

[57] Khurshid, S., Marinov, D., Shlyakhter, I., Jackson, D., *A Case for Efficient Solution Enumeration*, in SAT 2003, LNCS 2919, pp. 272–286.

[58] Khurshid S., Pasareanu C. S., Visser W. *Generalized Symbolic Execution for Model Checking and Testing*. In proceedings of TACAS'03, pp. 553–568.

[59] Lahiri S.K., Qadeer S., Galeotti J. P., Voung J. W. , and Wies T., *Intra-Module Inference*. In Computer Aided Verification (CAV '09), Springer Verlag, February 2009

[60] Leino. K.R.M. *Dafny: An Automatic Program Verifier for Functional Correctness*. In LPAR-16, volume 6355 of LNCS, pages 348-370. Springer, 2010.

[61] Leavens, G., Baker A., Ruby C. *Preliminary design of JML: a behavioural interface specification language for Java*. ACM Software Engineering Notes. Volume 31 Issue 3, May 2006.

[62] Leavens G., Poll E., Clifton C., Cheon Y., Ruby C., Cok D., Müller P., Kiniry J., Chalin P., and Zimmerman D. *JML Reference Manual (DRAFT)*, September 2009.

[63] Ma Y-S., Offutt J. and Kwon Y-R., *MuJava : An Automated Class Mutation System*, Journal of Software Testing, Verification and Reliability, 15(2):97-133, 2005.

[64] Marinov D., Khurshid S. *VAlloy: Virtual Functions Meet a Relational Language*. 11th International Symposium of Formal Methods Europe (FME), Copenhagen, Denmark. July 2002

[65] Mendonça de Moura L., Bjørner N. *Z3: An Efficient SMT Solver*. TACAS 2008, pp. 337–340.

[66] Meyer B. *Applying "design by contract"*. Computer, 25(10), pages 40–51. October 1992.

[67] Moskewicz, M., Madigan, C., Zhao Y., Zhang L., and Malik S. Chaff: engineering an efficient SAT solver. In J. Rabaey editor, *Proceedings of the 38th conference on Design automation*, pages 530–535, Las Vegas, Nevada, United States, 2001. ACM Press.

[68] Musuvathi M., Dill, D. L., *An Incremental Heap Canonicalization Algorithm*, in SPIN 2005: 28-42

[69] National Institute of Standards and Technology. *The economic impacts of inadequate infrastructure for software testing*, May 2002. `http://www.nist.gov/director/planning/upload/report04-2.pdf`.

[70] Near, J., Jackson D. *An Imperative Extension to Alloy*. 2nd Conference on ASM, Alloy, B, and Z (ABZ 2010). Orford, QC, Canada. February 2010.

[71] Robby, Dwyer M. B. , Hatcliff J. *Bogor: An extensible and highly-modular model checking framework*. In Proceedings of the 9th European Software Engineering Conference held jointly with the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2003.

[72] Pugh W. *The Omega test: A fast and practical integer programming algorithm for dependence analysis*. Communications of the ACM, 31(8), Aug. 1992

[73] Rugina, R., Rinard, M. C., *Symbolic bounds analysis of pointers, array indices, and accessed memory regions*, in PLDI 2000, pp. 182–195, 2000.

[74] S. Sagiv, T. W. Reps, R. Wilhelm. *Parametric shape analysis via 3-valued logic*. ACM TOPLAS 24(3): 217–298 (2002)

[75] Schulz S. *E: A Brainiac Theorem Prover* Communications, Volume 15(2/3) pp.111–126, 2002.

[76] Sharma R., Gligoric M., Arcuri A., Fraser G., Marinov D.*Testing Container Classes: Random or Systematic?*, in FASE 2011. March 2011, Saarbruecken, Germany.

[77] Shlyakhter I. *Generating effective symmetry breaking predicates for search problems*. Electronic Notes in Discrete Mathematics, 9, June 2001.

[78] Siddiqui, J. H., Khurshid, S., *An Empirical Study of Structural Constraint Solving Techniques*, in ICFEM 2009, LNCS 5885, 88–106, 2009.

[79] Spivey, J. M., *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, 1988.

[80] Taghiri M., *Automating Modular Program Verification by Refining Specifications*. PhD thesis. Massachusetts Institute of Technology, February 2008.

[81] Torlak E., Jackson, D., *Kodkod: A Relational Model Finder*. in TACAS '07, LNCS 4425, pp. 632–647.

[82] Torlak E., Vaziri M., Dolby J., *MemSAT: checking axiomatic specifications for memory models* In PLDI'10.

[83] Vallee-Rai R., Hendren L., Rajavallee-rai T.*Jimple: Simplifying Java Bytecode for Analyses and Transformations*, SABLE Research Group.Technical report. 1998.

[84] Vaziri, M., Jackson, D., *Checking Properties of Heap-Manipulating Procedures with a Constraint Solver*, in TACAS 2003, pp. 505-520.

[85] Visser W., Havelund K., Brat G., Park S. and Lerda F., *Model Checking Programs*, ASE Journal, Vol.10, N.2, 2003.

[86] Visser W., Păsăreanu C. S., Pelánek R., *Test Input Generation for Java Containers using State Matching*, in ISSTA 2006, pp. 37–48, 2006.

[87] Visser W., *Private communication*, February 2nd., 2010.

[88] Weidenbach C. *Combining superposition, sorts and splitting*. In Handbook of Automated Reasoning, volume II, chapter 27. Elsevier Science, 2001.

[89] Wing J. *Writing Larch interface language specifications*. ACM Transactions on Programming Languages and Systems, 9(1) pages 1–24, January 1987.

[90] Xie, Y., Aiken, A., *Saturn: A scalable framework for error detection using Boolean satisfiability*. in ACM TOPLAS, 29(3): (2007).

[91] Yessenov K. *A Light-weight Specification Language for Bounded Program Verification*. MIT MEng Thesis. May 2009.

Appendix A

DynJML Grammar

```
jDynAlloyModules      ::= jDynAlloyModule+

jDynAlloyModule       ::= 'module' id 'abstract'? (signature)?
                            (
                               jObjectInvariant
                               | jClassInvariant
                               | jObjectConstraint
                               | jRepresents
                               | jProgramDeclaration
                            )?

signature             ::= 'sig' id ( ('extends' |'in') id '' '' )? '' form? ''
jField                ::= 'field' id:type ';'

jClassInvariant       ::= 'class_invariant' form ';'
jObjectInvariant      ::= 'object_invariant' form ';'
jObjectConstraint     ::= 'object_invariant' form ';'
jRepresents           ::= 'represents' expr 'such_that' form ';'
jProgramDeclaration   ::= 'virtual'? 'program' id::id '[' jVariableDeclaration ( ',' jVariableDeclaration)*']'
                           'specification {'
                               jSpecCase*
                           '}'
                           'implementation {'
                               jBlock
                           '}'
```

jVariableDeclaration	::= 'var' id:type
jSpecCase	::= (jRequires | jModifies | jEnsures)*
jRequires	::= 'requires {' form '}'
jModifies	::= 'modifies {' expr '}'
jEnsures	::= 'ensures {' form '}'
jBlock	::= '{' jStatement+ '}'
jStatement	::=
	jAssert
	| jAssume
	| jVariableDeclarationStatement
	| jSkip
	| jIfThenElse
	| jCreateObject
	| jAssignment
	| jProgramCall
	| jWhile
	| jBlock
	| jHavoc
jAssert	::= 'assert' form ';'
jAssume	::= 'assume' form ';'
jVariableDeclarationStatement	::= jVariableDeclaration ';'
jSkip	::= ';'
jIfThenElse	::= 'if' form jBlock 'else' jBlock ';'
jCreateObject	::= 'createObject' '¡' id '¿' '[' id ']' ';'
jAssignment	::= expr ':=' expr ';'
jProgramCall	::= 'call' id '[' expr (',' expr)* ']' ';'
jWhile	::= 'while' form ('loop_invariant' form)? jBlock ';'

jHavoc	::= 'havoc' expr ';'	havoc
form	::=	
	expr 'implies' expr	implicacin
	| expr 'or' expr	disyuncin
	| expr 'and' expr	conjuncin
	| expr '=' expr	igualdad
	| not expr	negacin
	| id '[' expr (',' expr)* ']'	predicate
	| ('some' | 'all' | 'lone' | 'no' | 'one') (id ':' type)+ '{' form '}'	cuantificadores
	| 'callSpec' id '[' expr (',' expr)* ']'	callSpec
expr	::= id '[' expr (',' expr)* ']'	function
	| id	variable
	| number	literal Int
	| '(' expr ')'	
	| true — false	literal booleano
	| expr '+' expr	union
	| expr '+' expr	interseccin
	| expr '.' expr	join
	| expr '-¿' expr	producto
	| expr '++' expr	override
id	::= ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')* ""́?	
number	::= ('0'..'9')+;	