Profiling Alloy Models

by

Elias Eid

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

© Elias Eid 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modeling of software-intensive systems using formal declarative modeling languages offers a means of managing software complexity through the use of abstraction and early identification of correctness issues by formal analysis. Alloy is one such language used for modeling systems early in the development process. Nevertheless, little work has been done to study the styles and techniques commonly used in Alloy models.

We present the first static analysis study of Alloy models. We investigate research questions that examine a large corpus of 2,138 Alloy models. To evaluate these research questions, we create a methodology that leverages the power of ANTLR pattern matching and the query language XPath. We investigate the parse tree generated from each Alloy model and identify instances of formulated queries that are of interest to our research questions. We present the results and discuss the findings from examining these research questions.

Our research questions are split into three categories depending on their purpose and implementation complexity. Characteristics of Models include "surface-level" research questions that aim to identify *what* language constructs are used commonly. We also correlate certain model features using linear regression to determine the best predictors for model length and field count. Patterns of Use questions are considerably more complex and attempt to identify *how* modelers are using Alloy's constructs. Analysis Complexity questions explore the use of Alloy model features and constructs that may impact solving time.

We draw conclusions from the results of our research questions and present findings for language and tool designers, educators and optimization developers. Findings aimed at language and tool designers present ways to improve the Alloy language by adding constructs or removing unused ones based on trends identified in our corpus of models. Findings for educators are intended to highlight underutilized language constructs and features, and help student modelers avoid discouraged practices. Lastly, we present a number of findings for optimization developers that provide suggestions for back-end improvements.

Acknowledgements

I would like to thank my supervisor, Dr. Nancy A. Day, for her continued support and guidance throughout my graduate studies. Your expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I sincerely thank Dr. Joanne M. Atlee and Dr. Mei Nagappan, for their agreeing to read my thesis and for providing valuable feedback.

I would like to thank my colleagues Khadija Tariq, Tamjid Hossain, and Joseph Poremba for the role they played in helping me formulate research questions. I would also like to thank Ruomei Yan for her exquisite work on Catalyst. This thesis would not have been possible without your meaningful contributions.

I would also like to thank all of my professors, who through their extensive knowledge and dedication to teaching, helped me get to where I am today. As such, I thank my marvelous Computer Science and Mathematics professors including Dr. Rhonda Ficek, Dr. Daniel Brekke, Dr. Michael Haugrud and Dr. Bette G. Midgarden for always inspiring me and supporting me throughout my academic journey.

Finally, I would like to thank my family and friends for their continued love and support. I am forever grateful to my parents for their unconditional love and guidance. I would not be where I am today if it was not for their immense sacrifices and support.

Dedication

I dedicate my work to my family and many friends. A special feeling of gratitude to my loving parents, Hyam and Akram Eid, who gave up on so many of their dreams to help me pursue mine. When the times were tough, you were always there for me and I am forever grateful. This thesis stands as a testament to your unconditional love and encouragement. I also dedicate this thesis to my sisters Sandy and Léa whose words of encouragement and support ring in my ears. I dedicate this work and give special thanks to my best friend Gracia Hobeiche for being there for me throughout the entire graduate program.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software modeling is becoming an important part of the software development process in order to manage complexity and reduce development effort. Models are abstractions used to represent and evaluate the core elements of a system, devoid of superfluous design details. Software modeling helps developers identify and address bugs and design flaws early in the design phase before they make it to the implementation phase [31]. Models are often developed using specific techniques, tools and languages that can vary significantly in their breadth and formality. The Unified Modeling Language (UML) is one of the most successful and widespread modeling techniques that uses diagrams to model software entities and the relationships and constraints that bind them. As an object-oriented model, a UML diagram describes *how* the solution is obtained. While UML diagrams provide an intuitive abstract representation of a system, they are often not formal enough to permit the interactive property checking needed for the complex systems of today.

Unlike object-oriented modeling, declarative modeling describes *what* the solution should do. Declarative modeling allows developers to express the ideas, constructs and constraints of a software system succinctly at a high level of abstraction. The distinguishing feature of declarative modeling is that the system is described using constraints on abstract data usually expressed in first-order logic (FOL) and/or set theory. The models are not necessarily executable, but because they are formal, solvers can bring the models to life by finding instances and proving properties of the model. The sizes (scopes) of the sets are not fixed in the model but rather chosen for analysis. Many declarative languages have impressive texts and literature to learn the language (*e.g.,* [37, 57, 42, 18, 19, 50, 40]) and there are conferences dedicated to the paradigm (*e.g.,* the ABZ conference series [1]). There are also compilations of case studies or comparisons of modeling practices using these and related languages (*e.g.,* [23, 27, 51, 52]) and university courses that teach some of these

languages (*e.g.,* [44, 46]). Ball and Zorn [21] advocate for the importance of teaching software engineering students to model at this level of description. But little has been done to study empirically the state-of-the-practice in modeling using these languages.

Formal declarative modeling languages, such as Alloy [37, 38], TLA+ [42], B [18], Event-B [19], Z [50], VDM [40], and Abstract State Machines [22], are suitable for capturing structural and behavioral descriptions formally and abstractly in terms of sets, relations, and logical formulas that constrain the relationships. Because the model is formal, automated search and proof-based techniques provide the modeler with feedback regarding the correctness of the model early in the development process. Examples of the use of this level of modeling are: Zave's work using Alloy to discover problems in the CHORD protocol [59]; Newcombe *et al.*'s work with TLA+ at Amazon [45]; and Huynh *et al.*'s work with B on describing a healthcare access control model [35].

Our work aims to understand how people write Alloy models. We explore the characteristics of Alloy models as well as the patterns of use of the language. As declarative modeling becomes more popular and useful, it is important to examine how modelers use these languages in order to promote good practices (and acknowledge bad modeling practices), create teaching materials, and also offer suggestions for where analysis optimizations would be valuable because of common modeling practices. Our work also helps the designers of the language plan out future versions of the language that cater to the modelers' needs. Just learning a language is rarely enough to be able to use it well – we need to research the characteristics and patterns of models written in this paradigm.

We provide the first deep analysis of a corpus of Alloy models (2,138 models). We choose to investigate the use of the Alloy language because of its simplicity, wide-spread use, and openly accessible toolset called the Alloy Analyzer [4]. We present a variety of research questions to investigate common practices in these models. We determined these research questions from 1) existing literature on measures in programming and modeling, *e.g.,* set hierarchies and inheritance graphs, depth of quantifiers in formulas (*e.g.,* [25, 58]); 2) Alloy teaching material and discussions (*e.g.,* Jackson's Alloy book [37], Alloy Discourse [7], Stack Overflow [5]), and 3) interactions with others in our research group investigating Alloy modeling and tools (*e.g.,* [30, 20, 41]).

We divide our research questions into three categories: 1) Characteristics of Models; 2) Patterns of Use within Models; and 3) Analysis Complexity. Characteristics of Models cover "surface-level" research questions that aim to identify *what* language constructs are used frequently. We also correlate certain Alloy model features using linear regression to produce tangible results that determine the best predictors for model length and field count. Patterns of Use questions attempt to identify *how* the language constructs are used

and consequently are significantly more involved than the questions in Characteristics of Models. Lastly, the research questions in Analysis Complexity explore the use of model features and constructs that impact solving time. For each research question, we provide motivation, our approach to answering the question, the results and a series of findings aimed at language and tool designers, educators and optimization developers.

1.1 Contributions

The main contributions of our work are:

1. Motivation for research questions relevant to Alloy modeling.
2. A methodology for answering these research questions on Alloy models using static analysis.
3. Evaluation of these research questions on a large corpus of Alloy models.
4. A series of findings aimed at language and tool designers to help them evolve the Alloy language and its tool support based on modelers' use patterns.
5. A series of findings aimed at educators that highlight underutilized constructs and language features in addition to bad modeling practices that student modelers should avoid.
6. A series of findings aimed at optimization developers that suggest future optimization techniques for analysis of models in the Alloy language.

1.2 Thesis Outline

We start by providing a brief background about the Alloy language in Chapter 2. In Chapter 3, we introduce our methodology for static analysis of Alloy models. Chapter 4 discusses the research questions that fall under the Characteristics of Models category. We describe the research questions (RQ), approach, results and findings for evaluating these research questions on the corpus of Alloy models. Similarly, Chapter 5 contains the Patterns of Use research questions. In Chapter 6, we investigate the Alloy language features that may affect analysis complexity and solving time. We also discuss future optimizations that can be added to the different Alloy solvers. Chapter 7 presents related work and Chapter 8 provides a summary of the work presented and concluding remarks.

Chapter 2

Background

In this chapter, we provide a brief introduction to the Alloy language along with an overview of the Alloy Analyzer, a tool used to create and analyze Alloy models. We also discuss the constructs of the language used to express the structure and behavior of the model.

2.1 The Alloy Analyzer

The Alloy Analyzer is used to build, edit and analyze Alloy models via its Kodkod engine [53] and SAT solvers. Typically, Alloy models contain command queries that are questions about the model asked by the user. The Analyzer checks the command queries of the model for finite sizes of the sets. If a query is unsatisfiable, the Alloy Analyzer produces a message informing the modeler that no satisfying interpretation could be found. When a model is satisfiable, the Alloy Analyzer produces instances *i.e.,* interpretations that satisfy all the constraints expressed in the model. The Alloy Analyzer is a GUI application with three main components:

- An editing interface for modifying system specifications.
- A solutions display to showcase potential instances that the Analyzer discovered.
- A section containing statistics about the internal data structures employed during the analysis.

The two main commands for searching using the Alloy Analyzer are `run` and `check`, each of which can be associated with a set of constraints called a specification. The

`run` command tells the Analyzer to find an example that satisfies all of the constraints of the given specification and the model, basically providing a satisfying instance of the specification. When the `check` command is given by the user, the Analyzer searches for a counterexample, *i.e.,* an instance that satisfies the model's constraints but not the specification, which would refute the specification's correctness. The Alloy Analyzer can also display solutions in a graphical format that uses nodes to represent atoms and lines and arrows to represent the relationships between them.

2.2 Alloy

Alloy [38] is a modeling language that can express the fundamental structure and behavior of a system in addition to the constraints and operations that dictate how the system may change. The Alloy language combines relational calculus and first-order logic with the transitive closure and set cardinality operators and limited support for arithmetic. Statements in Alloy are expressed using ASCII text characters. The language was originally developed by Daniel Jackson at MIT's Software design group [36]. Jackson coined Alloy to address the shortcoming of the Z notation [24] such as excessive notation and the lack of more recent constructs used in object models. The Alloy language contains a relatively small number of constructs making it an easy language to learn and analyze. The Alloy language is composed of a variety of packaging constructs:

1. **Signatures**: introduce a new set of atoms. Signatures may be declared with a multiplicity. Multiplicity keywords like `one`, `some` and `lone` specify the size of a signature. Signatures declared with multiplicity `one` have exactly one element in them. Signatures declared with multiplicity `some` have at least one element in them. Lastly, signatures declared with multiplicity `lone` have zero or one element in them.

   Signature declarations may contain fields. **Fields** are written in the body of signatures and dictate how signatures are connected to each other. For instance, the following signature declaration `sig A {f1: B}` introduces a new field `f1` whose domain is `A` and whose range is `B`. The declared set is always the first argument of a field *e.g.,* `sig C {f2: A -> B}` creates a field between `C`, `A` and `B`. We use the term **signature** to refer to the unary set introduced by a signature declaration (*e.g.,* the signature `A`).

   There are constructs to declare a set to be a subset or extension of another set, creating a set hierarchy in the model. **Subsignature extensions** are mutually disjoint subsets of a parent signature introduced using the keyword `extends`. **Subset**

signatures are inclusive subsets of a parent signature declared using the keyword `in`.

Signatures may include formulas associated with the declared set. The block containing formulas in a signature declaration is known as a **signature fact block**.

2. **Formulas**: denote constraints on sets and fields expressed in Alloy's logic, thus limiting the possible values of the sets and fields of the model. Formulas can be grouped within a fact block (including signature fact blocks), an assertion block, a predicate, a function or a macro.

3. **Predicates**: are constraint containers that can be used elsewhere in the model. Predicates may be parameterized *i.e.,* they can include zero or more arguments with explicitly set types. Predicates allow the model to be modular as they can be included in the analysis only when needed and can be reused in different contexts.

4. **Functions**: are named expressions that return a value. Functions may be parameterized. Function parameters need to have explicit types. Functions can be called in formulas.

5. **Macros**: are defined using the `let` keyword at the top level of a file. Unlike predicates and functions, macros are untyped *i.e.,* parameters do not need to be given types.

6. **Assertions**: are a set of formulas. Assertions should follow from the facts of the model and and can be checked in a command. Unlike predicates and functions, assertions do not take parameters. Assertions are remnants of an older version of Alloy and can be essentially thought of as unparameterized predicates [9].

7. **Commands** (also called **queries** or **command queries**): denote questions that a user asks about the model. `run` commands check whether there is an instance of the model that satisfies all the constraints and are used with predicates and functions. `check` commands search for a counterexample to a constraint. Commands can be supplemented with **scopes** that limit the size of the signature sets in instances or counterexamples that will be considered.

Throughout this work, we use the term **signature declaration** to refer to the entire packaging construct that contains the signature declaration along with the fields and any formulas. Models in Alloy are composed of statements created using a combination of these constructs. The order of statements does not affect the model since Alloy is a declarative

language. An Alloy model may be partitioned into multiple files. The subfiles of a model are usually called **modules** and can be imported into a model using an `open` statement. We will explain the constructs of the Alloy language in more detail with examples as needed in the chapters that follow.

2.3 Types and Type Checking

The type system in Alloy has two main functions. First, a rudimentary type checking mechanism allows the Alloy Analyzer to catch errors before analysis is performed. Ill-typed expressions in Alloy are expressions that can be shown to be redundant using types alone. Second, the type system in Alloy is used to resolve overloading *e.g.,* if two signatures have fields with the same names, the type of an expression enables the Alloy Analyzer to determine which signature and field are referenced. Alloy has two kinds of types:

- **Basic Types**: these types are implicitly associated with signatures. Each top-level signature in a model is assigned a unique type. Subsignature extensions are also given unique types. However, subset signatures are not given their own type but acquire the parent signature's type instead. Two types overlap if one type is a subtype of the other *i.e.,* the subtype is associated with a subsignature extension of another signature with a different type.
- **Relational Types**: every expression in an Alloy model is assigned a relational type consisting of a union of products. Each product term in the union must have as many basic types as the arity of the relation.

```
1  sig A {f1: B}
2  sig B {}
3  pred some_f_union_B {some f1 + B}
4
5  sig C {}
6  sig D {f2: set C}
7  sig E extends B {f3: C}
```

Figure 2.1: Sample Alloy Model

The Alloy Analyzer can identify two kinds of type errors. The arity of a field is computed by counting the sets in the type expression of a field declaration plus one for the

signature under which the field is declared (*e.g.,* `f` in `sig A { f : x -> y }` has an arity of 3). The first kind of type error arises when the modeler attempts to form expression of mixed arity. We will use the signatures and predicates in Figure 2.1 to provide example of type errors. For example, the expression `f1 + B` in the body of the predicate `some_f_union_B` is an illegal expression, since `f1` has arity two and `B` has arity one. The second kind of type error occurs when an expression is equivalent to or contains the empty set. For instance, `E.f2` is a redundant expression since `f2` maps elements of basic type `D` and no element in `E` is also in `D` so the expression always results in the empty set. Similarly, `(D + E).f2` is also an erroneous expression because elements of basic type `E` cannot be mapped by `f2` and thus this expression contains the empty set and can be written equivalently as `D.f2`.

Chapter 3

Methodology

In this chapter, we discuss the corpus of models used in this study and provide an overview of our static analysis methodology. We explore the tools used to extract elements and patterns from Alloy models as well as the statistics generated to answer our research questions. Lastly, we provide an overview of the linear regression methodology used to correlate model features in subsequent chapters.

3.1 Corpus of Models

Our goal is to survey a diverse set of Alloy models to answer our research questions. To build our corpus of Alloy models, we use Catalyst, a tool developed by our research group [14] for scraping Alloy models from github repositories. The tool uses standard techniques to gather publicly available Alloy models, including the ones available with the Alloy Analyzer [4], ones scraped from public github repositories and other sources (*e.g.,* the Platinum evaluation models [60] and the 56 models provided in Jackson's book on Alloy [37]). We ensure there are no files that are exact duplicates of each other in this corpus which includes replicas of the models in Jackson's book on Alloy that could have been created by student modelers attempting to learn the language. We also remove any library models that are part of the Alloy language/Analyzer. We exclude files that do not parse correctly with the Alloy Analyzer, which ensures that all models conform to the Alloy well-formedness constraints.

Next, we filter this corpus to ensure diversity of models because multiple versions of the same model may appear in a repository. For repositories that contain iterative versions of

the same model, we choose the "highest" version of models to represent the most advanced model when possible. In total, our corpus contains 2,138 Alloy files which includes models drawn from 503 different github repositories. Within these, there are 31 models created by our research group prior to this work.

In our corpus, we assume that these models are in a mostly complete state. We do not know how many distinct modelers are the authors of these models. We also do not know the purpose of these models *i.e.,* we do not know if they are used in industry or for educational purposes. Our corpus may contain automatically generated models which means that one person's modeling style and preferences could be affecting many models.

3.2 Static Analysis

To answer our research questions, we statically analyze textual Alloy files. For each research question, we create one or more queries, search for instances of the query(-ies) in the file, and then collate the results across the file and/or multiple files. Figure 3.1 is an outline of our methodology.

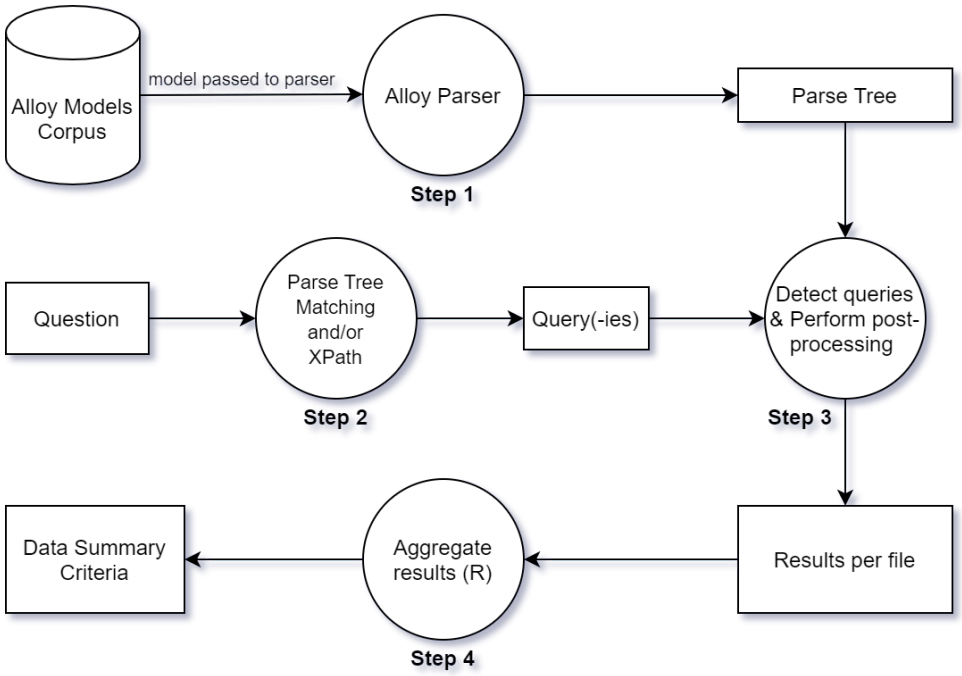

Figure 3.1: Methodology

First, we create and test an ANTLR [47] grammar for Alloy as shown in **Step 1** of Figure 3.1. The complete Alloy grammar is presented in Appendix A. Our ANTLR parser accepts models that are written in the input languages for the Alloy Analyzer versions 3 - 5. The differences between these languages are very small. There are no syntactic changes in the language between versions 4 and 5 [13]. We opt to create our own ANTLR parser for Alloy as opposed to using the existing Alloy grammar written in CUP [2] because ANTLR offers built-in pattern matching tools and support for the query language XPath [55]. We create a complete Alloy grammar using a combination of lexer (terminal) rules and parser (non-terminal) rules. **Terminal** symbols are the elementary symbols of the grammar that cannot be decomposed using the rules of the grammar. Terminal rules in ANTLR are fixed strings or can be defined using regular expressions. **Non-terminals** are the symbols in the grammar that are composed of a combination of terminals and non-terminal symbols[1]. Consider the example in Figure 3.2 that shows an excerpt of the Alloy grammar. In the signature declaration grammar rule, `priv`, `abs`, `multiplicity`, `names`, `sigExtention`, `decls` and `block_opt` are examples of non-terminals, whereas `'sig'` and `'{' '}'` are terminal symbols defined using fixed strings. `ALPHA`, `DIGIT` and `ID` are terminal rules defined using regular expressions. Our ANTLR parser generates a parse tree from any syntactically well-formed Alloy model.

```
1  paragraph : factDecl | assertDecl | funDecl | cmdDecl |
       enumDecl | sigDecl | predDecl;
2
3  sigDecl : priv abs_multiplicity 'sig' names sigExtension '{'
       decls '}' block_opt;
4  name : ('this/')? (ID '/')* ID;
5  names: name (',' name)*;
6
7  ALPHA: [a-zA-Z_"]+ ;
8  DIGIT : [0-9] ;
9  ID : ALPHA ( ALPHA | DIGIT )* ;
```

Figure 3.2: Excerpt of Alloy Grammar

ANTLR parsers can generate parse trees or abstract syntax trees from an input. We examine the parse tree of the model (rather than the abstract syntax tree) because it contains all the information from the file including the whole string associated with each

[1]Definitions of terminals and non-terminals vary in the literature.

non-terminal and terminal, and it allows for seamless extraction of subtrees and nodes. Abstract syntax trees nodes only contain the type information of each node *i.e.,* each node is labeled with the terminal or non-terminal rule it pertains to and does not contain the string literal value associated with it. Abstract syntax trees also require user-generated visitors to traverse whereas parse trees allow for easy retrieval of the literal string values associated with each node via external libraries and ANTLR's built-in tools which we will discuss shortly. Our methodology identifies and collects exact matches from the models. Consequently, working with parse trees allows us to generate powerful yet concise scripts that extract all the required information needed for our exhaustive profiling of Alloy models.

Our **research questions** vary greatly in complexity and require a wide range of extracted information from the model. We use the query language XPath (and its libraries) in addition to ANTLR's built-in parse tree matching to create **queries** and extract information from the Alloy models as shown in **Step 2** of Figure 3.1. We first describe the use of the XPath query language and then we discuss ANTLR's built parse tree matching mechanism.

To search for instances of a subtree in an Alloy file, we use the query language XPath. Originally, XPath was a query language for selecting nodes in XML documents. Support for XPath was added to the parser-generator ANTLR with its version 4.

Separator	**Description**
nodename	Nodes with the token or rule name "nodename"
/	All **direct** descendants that match the next element in the path. Selection starts at the root node if used at the start of the hierarchy path
//	All descendants in the tree that match the next element in the path. Selection occurs anywhere in the tree if used at the start of the hierarchy path
!	Any node except the next element in the path
*	Any node in the path

Table 3.1: XPath Separators

An XPath **hierarchy path** is a sequence of expressions describing a hierarchy in the parse tree. Each expression is a non-terminal or terminal node in the grammar or a combination of expressions and separators. Table 3.1 provides a list of all the XPath separators along with their definitions. Separators describe quantification over the nodes to match. When an XPath hierarchy path begins with a "/" operator, the selection occurs at the root node. The "//" operator at the start of an XPath hierarchy path indicates that

the selection can begin anywhere in the parse tree. The any ("*") and not ("!") separators are combined with the "//" or "/" separators to indicate if they apply to all descendants of the previous expression or just the direct descendants respectively. The "not" operator is a negative operator, meaning it chooses nodes that do not match the specified node type in the string path. XPath can extract subtrees from the model parse tree. Subtrees can consist of any number of nodes. A subtree containing only one node corresponds to a terminal rule in the grammar. A subtree with two or more nodes always has a root node that corresponds to a non-terminal rule in the grammar. Consider the sample parse tree shown in Figure 3.3. The XPath hierarchy path `"/A/B"` extracts the subtree consisting only of the B node at level 1 since it is a direct descendant of A. `"/A//B"` extracts the two B nodes (at levels 1 and 2) from the tree since they are descendants (not necessarily direct) of A. The following hierarchy path `"/A/*"` extracts any direct descendant of A (*i.e.,* the B node and the C subtree at level 1). `"/A/!B"` extracts all direct descendants of A that are not of B nodes (*i.e.,* just the C subtree at level 1).

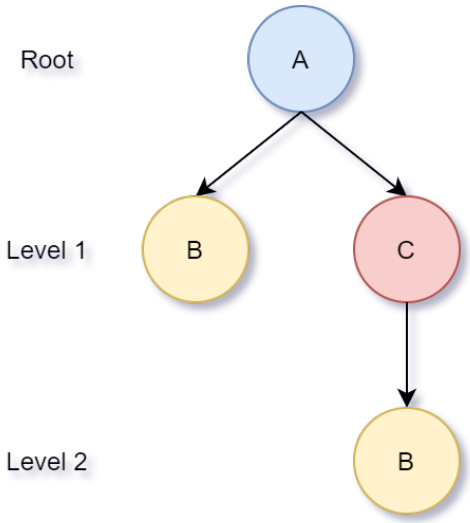

Figure 3.3: Sample Parse Tree

An XPath hierarchy path is sufficient when the research question requires extracting one kind of subtree from the parse tree. For instance, we can identify all signature names using the following XPath hierarchy path

```
"//sigDecl/names/name"
```

that extracts all name subtrees located in signature declarations. The hierarchy path is passed to the XPath `findAll` method along with the model parse tree and the parser:

```
Collection<ParseTree> sigNames = XPath.findAll(tree,
        "//sigDecl/names/name", parser);
```

The method call returns a `Collection` of name subtrees.

When a research question requires extracting a subset of node kind or a subtree that conforms to a particular pattern that cannot be expressed in an XPath hierarchy path, we use ANTLR's built-in parse tree matching (along with XPath to narrow the search space). A **parse tree pattern** is a string that describes what we want to match in the file. It can contain terminals, non-terminals and strings from the grammar. Strings from the grammar correspond to the literal value of certain terminal rules. For instance, "`State`" is a possible literal value of the terminal identifier rule (`ID`). Parse tree matching allows us to define a pattern containing the literal value "`State`" that is converted to a parse tree and then matched against the entire model's parse tree or a portion of it.

We use XPath hierarchy paths to select a subset of the parse tree to narrow the search space for the pattern. When combined with parse tree matching, XPath is not used extract subtrees, instead we use it to extract a subset of the model parse tree and then we use parse tree matching to identify a subset of trees that conform to a pattern. For instance, we must use a parse tree pattern to extract signature declarations with multiplicity `one` given that an XPath hierarchy path is only capable of extracting all signature declarations regardless of multiplicity. The following parse tree pattern

```
"<priv> one sig <names> <sigExtension> { <decls> } <block_opt>"
```

represents an Alloy signature declaration with multiplicity `one`. The parse tree pattern can be created as follows:

```
ParseTreePattern p = parser.compileParseTreePattern("<priv>
  one sig <names> <sigExtension> { <decls> } <block_opt>",
                  ALLOYParser.RULE_sigDecl);
```

The parse tree pattern is then passed to ANTLR's built-in `findAll` method along with the model parse tree and the XPath hierarchy path `"//paragraph/*"` that limits the search space to the direct descendants of paragraph trees:

```
List<ParseTreeMatch> matches = p.findAll(tree,
                    "//paragraph/*");
```

The method call returns a list of `ParseTreeMatches` that can be decomposed further or converted to parse trees.

Once the **query** (*i.e.,* XPath string and/or parse tree pattern) for a research question has been formulated, we detect and extract instances of the query from the model as shown

in **Step 3** of Figure 3.1. After instances of the pattern have been extracted, some queries require a post-processing step to refine the data, such as handling multiple elements within one string and any calculations per file. In some cases, the constructs relevant to a research question can include related forms, which require the use of multiple parse tree patterns and occasionally multiple XPath hierarchy paths. Negative patterns are also used to identify instances that do not conform to a certain construct. Sometimes, the parse tree must be traversed multiple times to determine correctly the answer to the query (*e.g.,* integer variables have to be identified before we can determine how they are used). For some research questions, the count of a particular construct in a model is scaled according to the number of calls made to predicates and functions containing instances of this construct. We discuss scaling construct counts in more depth as needed in the next chapters. By leveraging the flexibility of XPath and the profuseness of parse tree data, we create a versatile methodology for identifying various patterns in Alloy models.

The data resulting from multiple Alloy model files is combined using an R script [6] as shown in **Step 4** Figure 3.1. For each research question, we find some or all of the following data summary criteria to be of interest:

- **Predominant Use (PU)**: The **mode** per file identifies the most recurrent value in the set of collected values and thus identifies the most frequent form/use of each pattern.
- **Typical Use (TU)**: The **median** per file provides the middle value in the sorted list of data points. We opted for the median as a measure of central tendency as opposed to the mean because our generated data is often heavily skewed and contains several outliers. The typical use criterion provides an aggregated value that summarizes the data set without running the risk of being skewed by outliers.
- **Distribution (D)**: The **percentage distribution** as measured across all files (or occurrences) is used when the goal of the research question is to identify the partition of a data set into a number of categories.
- **Percentile Distribution**: A percentile is a value below which a percentage of values in the data set fall. We provide a percentile distribution that includes the 12.5^{th}, 25^{th}, 50^{th}, 75^{th} and 87.5^{th} percentiles.
- **Common Range (CR)**: We define the common range as the range that encompasses 75% of values in the data set *i.e.,* the values that fall between the 12.5^{th} percentile and the 87.5^{th} percentile.

In our results, values with an asterisk (*) indicate a non-zero criterion *i.e.,* zeros were eliminated from the data before computing the value.

3.3 Linear Regression

Linear regression is a statistical tool used for predictive analysis. Linear regression models the relationship between two variables by attempting to find the best linear model (*i.e.,* line) that fits the data. The methodology we use to perform linear regression is the same one employed by Lopes and Ossher in [43]. We use linear regression to correlate certain model characteristics in Chapter 4. Linear regression is often conducted and plotted in the linear scale. However, certain transformations can be applied to the data to address issues such as skewness and outliers. One such transformation is the logarithmic transformation used to address skewness in the data. The logarithmic transformation is applied by taking the natural logarithm of both variables. Data produced from examining software artifacts is often heavily skewed and the data generated from our corpus of Alloy models is no different. Thus, we opted to use the logarithmic transformation for all variables in our linear regression analysis. Figure 3.4 and Figure 3.5 show the signature count in Alloy models plotted in linear and logarithmic scale respectively. Figure 3.4 clearly shows that the data is heavily skewed given that most data points have small values for the length. After applying the logarithmic transformation, we can see an almost perfect log-normal distribution as shown in Figure 3.5. The logarithmic transformation can only be applied to data sets that do not contain zeros. Therefore, for our research questions we take several measures to ensure that the produced data sets do not contain zeroes. We discuss these measures in detail as needed in Chapter 4.

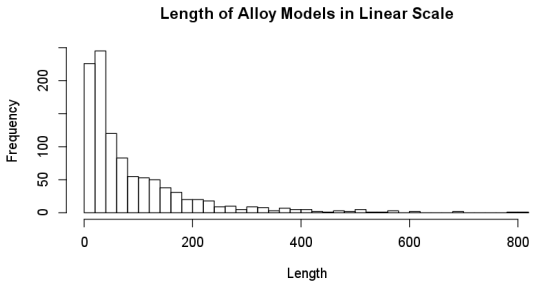

Figure 3.4: Histogram of Model Length Plotted in Linear Scale

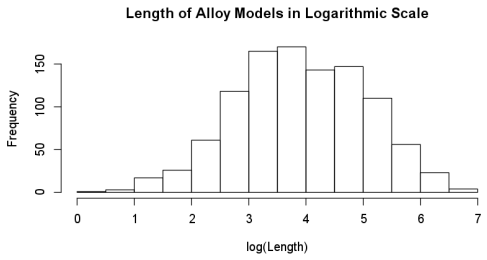

Figure 3.5: Histogram of Model Length Plotted in Logarithmic Scale

In both linear and log scales, the best fit line is given as $y\_values = \alpha + \beta x\_values$. However, in a log scale plot, the best fit line represents $log(y) = \alpha + \beta log(x)$. When we transform this equation back into the linear scale, we get the following equation that describes the exponential relationship between the two variables:

$$y = e^{\alpha} x^{\beta} \tag{3.1}$$

The exponential relationship between the two variables X and Y is dictated by the value of β as follows:

- If $\beta = 1$, the relation between the variables X and Y reverts to linear.
- If $\beta \neq 1$, then there exists an exponential relation between X and Y .
- If $\beta > 1$, then the relationship between X and Y is superlinear *i.e.,* Y grows exponentially faster as X grows.
- If $\beta < 1$, then the relationship between X and Y is sublinear *i.e.,* Y grows exponentially slower as X grows.

The **correlation coefficient** r measures the strength and the direction of a linear relationship between two variables on a scatter plot. The value of r is such that $-1 \leq r \leq +1$. The polarity of r indicates the kind of correlation that exists between the two variables as follows:

- **Positive Correlation**: as the values of X increase, the values of Y increase as well. If X and Y have a strong positive correlation, then r will be positive such that $r > +0.8$. An r value of exactly $+1$ indicates a perfect positive fit.
- **Negative Correlation**: as the values of X increase, the values of Y decrease. If X and Y have a strong negative correlation, then r will be negative such that $r > -0.8$. An r value of exactly -1 indicates a perfect negative fit.
- **No Correlation**: if no correlation exists between the two variables X and Y, then the value of r will be zero or very close to zero.

Table 3.2 shows the interpretation guidelines[2] for the correlation coefficient r proposed by Hinkle *et al.* in [33].

[2]These guidelines differ in the literature depending on the data characteristics and the purpose of the study

Correlation Strength	Positive	Negative
Negligible	0.0 to 0.3	-0.0 to -0.3
Low	0.3 to 0.5	-0.3 to -0.5
Moderate	0.5 to 0.7	-0.5 to -0.7
High	0.7 to 0.9	-0.7 to -0.9
Very High	0.9 to 1.0	-0.9 to -1.0

Table 3.2: Guidelines for Interpretation of Correlation Coefficient r

3.3.1 Goodness of Fit

The correlation coefficient r reflects the relationship between two variables but it does not reflect the percentage of values that exhibit this correlation. Examining the **goodness of fit** is a critical part of linear regression. R^2 , also known as the **coefficient of determination**, is statistic measure used to determine the proportion of variance in the dependent variable that can be explained by the independent variable *i.e.,* R^2 shows how well the data fits the regression model. For instance, if $R^2 = 0.95$, then 95% of the variation in the data can be explained by the best fit line. The coefficient of determination R^2 is the square of the correlation coefficient r .

While R^2 is often sufficient to determine the goodness of fit, its predictive value can be limited in some cases where the data exhibits certain characteristics. Therefore, it is always important to examine the *residuals* of the regression model. A **residual** is the vertical distance between any one data point and its estimated value. We typically consider four plots involving residuals: Residuals vs. Fitted, Normal Q-Q, Scale-Location and Residuals vs. Leverage. If the data fits the regression model, these plots will feature the following characteristics:

- **Residuals vs. Fitted**: is a scatter plot of residuals on the y axis and fitted (*i.e.,* estimated) values on the x axis. If the regression model is a good fit, then the Residuals vs. Fitted plot should show randomly distributed data around a horizontal line at the origin which indicates that what is left from the fit is unbiased noise. The line at the origin may not necessarily be a perfectly horizontal line. The fit is acceptable as long the line does not show a clearly discernible pattern (*e.g.,* a parabola).

- **Normal Q-Q**: is a scatter plot created by plotting two sets of quantiles (also known as percentiles) against one another. If the points form an approximately straight

line, then the residuals fit a normal distribution which is the ideal case. However, in practice some deviation is to be expected in the Q-Q plot especially towards the ends. Thus, light-tailed Q-Q plots are acceptable when performing linear regression.

- **Scale-Location**: also known as Spread-Location, this plot is very similar to the Residuals vs. Fitted plot. However, it takes the square root of the absolute value of standardized residuals instead of plotting the residuals themselves. The Scale-Location plots shows if residuals are spread equally along the ranges of predictors. It is used to check the assumption of equal variance (homoscedasticity). Ideally, we would like to have a horizontal line with equally spread points on both sides. Nevertheless, horizontal lines that are lightly-tailed towards the ends are still acceptable.

- **Residuals vs. Leverage**: this plot shows the leverage or influence that data points have on the fit. It is used to identify influential outliers in the data, if any. Unlike the previous plots, we do not look for trends in the graph, instead we look for cases that fall outside Crook's distance lines (plotted as dashed lines). If there are cases outside Crook's distance lines, then these data points have an undue influence on the regression model which will be altered when they are excluded.

3.4 Summary

In this chapter, we discuss the origin of the models used in our corpus study, the methodology employed to profile the models and the statistical tools used to formulate results. Our corpus contains 2,138 Alloy models, most of which where scraped from public github repositories. We apply different filters to the corpus to ensure that the models are diverse. We perform our static analysis of Alloy models using ANTLR's built-in parse tree matching mechanism and the query language XPath. We use XPath to extract from the parse tree subtrees of the same kind whereas parse tree matching is used for more intricate patterns that cannot be expressed using an XPath hierarchy path. For each research question, we generate one or more of the following data summary criteria: predominant use (mode), typical use (median), percentage distribution, percentile distribution and common range. We use linear regression to correlate certain Alloy model features. We apply the logarithmic transformation to our data sets which are often heavily skewed. The correlation coefficient r determines the strength and polarity of the correlation between the variables. The coefficient of determination R^2 is a measure of the goodness of fit of the model and reflects the percentage of data points that can be explained by the correlation equation. We also examine the residual plots to ensure that the regression model is a good fit.

Chapter 4

Characteristics of Models

In this chapter, we investigate research questions that investigate which language constructs modelers are using in their Alloy models. The research questions in this chapter are "surface-level" questions that explore the superficial characteristics of Alloy models through the use of relatively simple patterns that require a single pass through the parse tree. The post-processing in these questions, if any, is fairly minimal and does not make use of external data structures. By learning about the constructs most commonly used in Alloy models, language and tool designers can concentrate their efforts on improving these constructs. Educators can focus their attention on teaching these constructs.

4.1 Model Length

Alloy models often seem considerably shorter than programs. In this section, we explore the length of Alloy models by measuring the number of lines.

RQ# 1: How long is the average Alloy model?

Motivation: The simplest characteristic of any model is its length. By measuring the length of Alloy models, we can get a better understanding of how they compare to programs in terms of length.

Approach: We count lines in each Alloy model (not including blank lines and comments) and report the predominant and typical use criteria, which represent the most frequent model length and the central tendency of the model length respectively.

Results: We find that the predominant value (*i.e.,* mode) for model length is 52 lines,

whereas the typical value (*i.e.,* median) is 62.

Findings: We find that:

- Alloy models are fairly short especially when compared to programs.

In Section 4.5, we correlate model length with set and formula counts to get a better understanding of the model characteristics that have the most significant effect on length.

4.2 Signatures

In this section, we discuss research questions related to signature use in Alloy models. We attempt to quantify the frequency of signature use in Alloy models and identify the kinds of signatures that are most prevalent. We also explore the use of signatures as scalars.

```
1  sig A {
2   f1 : B1    // relation f1: A -> B1
3   f2 : A     // relation f2: A -> A
4  }
5  abstract sig B {}
6  one sig B1 in B {
7   f3, f4 : C1     // relations f3: B1 -> C1, f4: B1 -> C1
8  }
9  sig A1, A2 extends A {}
10 enum C {C1, C2}
```

Figure 4.1: Alloy Signature Declarations Example

RQ# 2: How often do modelers use signatures in Alloy models?

Motivation: Signatures are a fundamental component of any Alloy model since Alloy does not have scalars. **Signatures** are used to introduce new sets (including subsets and extensions). Figure 4.1 shows a small example of Alloy signature declarations. Five sets are introduced using signature declarations in this example (`A`,`A1`,`A2`,`B`,`B1`). The `enum` declaration on line 10 introduces three additional sets (`C`, `C1`, `C2`). By examining signature and `enum` declarations we can determine the profuseness of sets in Alloy models.

Approach: We extract all signature declarations as well as sets introduced in `enum` declarations from a model and perform post-processing on the collected matches to account for

multiple signatures aggregated in one declaration. For instance, the signature declaration on line 9 introduces two sets `A1` and `A2`. We choose to compute the predominant use and typical use criteria as they offer a comprehensive answer to the question. The predominant use criterion computes the most recurrent number of signatures per file across all the Alloy models. The typical use criterion aggregates the results by computing the central signature count value.

Construct	**Predominant Use** (Mode)	**Typical Use** (Median)
Signatures	2	8

Table 4.1: Use of Signatures

Construct	**12.5th**	**25th**	**50th**	**75th**	**87.5th**
Signatures	2	3	8	20	32

Common Range: $[2, 32]$

Table 4.2: Percentile Distribution of Signature Count

Results: The results of this research question are shown in Table 4.1 and Table 4.2. The typical use value of eight indicates that a typical Alloy model contains eight signatures, although the predominant use value is two. The data generated for this query is heavily skewed which shows that signature count in Alloy models exhibits a great deal of variation. The skeweness of the generated data is evident when examining the percentile distribution in Table 4.2 which shows a significant level of disparity between successive percentiles. The common range spans a cross a wide range ($[2, 32]$) which means that 75% of signature counts fall between one and twenty.

Findings: Based on our results, we conclude that:

- The signature count in Alloy models varies significantly between one model and another.

RQ# 3: How often are modelers using top-level signatures, subset signatures and subsignature extensions?

Motivation: The Alloy language contains different kinds of signatures. **Top-level signatures** create parent sets that are not subsets of another set (*e.g.,* lines 1 and 5 of

Figure 4.1). Top-level signatures are mutually disjoint. **Subset signatures** are declared as subsets of another signature using the keyword `in` (*e.g.,* set `B1` is a subset of `B` on line 6 of Figure 4.1). Subset signatures are not necessarily mutually disjoint unless they are explicitly constrained to be in a formula. **Subsignature extensions** create mutually disjoint subsets of a set and are declared using the keyword `extends` (*e.g.,* sets `A1`, `A2` on line 9 of Figure 4.1 extend set `A`). Signature extensions can also be introduced using `enum` (*e.g.,* `C1` and `C2` are extensions of the top-level signature `C`). By examining the kinds of signatures used in Alloy models, we can get a better understanding of the set hierarchies used by modelers, which can help us determine if a more advanced type checking system should be added to the Alloy language.

Approach: We extract and tally up the number of top-level signatures, subset signatures and subsignature extensions in each model. We account for subsignature extensions introduced using `enum` when tallying up the number of extension signatures.

Signature	PU	TU	D
Top-level	3*	3	16.5%
Subset	1*	0	1.2%
Extension	2*	4	82.3%
			} = 100%

Table 4.3: Signatures by Level

Results: We find that top-level signatures account for 16.5% of all signatures. Subsignature extensions are the most prominent kind of signatures coming in at 82.3%. Subset signatures are quite sparse in Alloy models (1.2% of all signatures). The typical Alloy model contains three top-level signatures and four subsignature extensions but no subset signatures.

Findings: We find that:

- Given the prominence of subsignature extensions, we conclude that the most common use of set hierarchy is partitioning the universe.
- Types are commonly used in other languages to partition a universe of atoms. Types also allow for type checking. We suggest that type checking mechanisms be explored for the Alloy language to provide faster feedback to users.

RQ# 4: How often do modelers use `abstract` signatures?

Motivation: An **abstract signature** has no elements except those belonging to its extensions or subsets (*e.g.,* set `B` on line 5 of Figure 4.1). The number of `abstract` signatures

in a model gives us an insight into its use of inheritance since most `abstract` signatures exist for the sole purpose of creating extensions from them. Beyond inheritance, `abstract` signatures offer many advantages such as enabling modelers to take advantage of Alloy's scope inference mechanism and allowing modelers to create more concise models that are easier to evolve in the future. We discuss the advantages of `abstract` signatures in greater detail in Chapter 5.

Approach: We extract signature declarations containing the keyword `abstract`. The predominant use criterion represents the most frequent number of `abstract` signatures per model, whereas the typical use criterion provides the median number of `abstract` signatures over all the collected files. We also present the percentage distribution of `abstract` signatures out of the total number of signatures.

Results: We find that `abstract` signatures are fairly uncommon in Alloy model. The predominant use criterion is zero (non-zero PU is one) whereas the typical use criterion is one. Thus, the typical Alloy model contains only one `abstract` signature. We also find that `abstract` signatures account for 8.1% of the total number of signatures across all models.

Findings: The findings of this research are as follows:

- `abstract` signatures are used sparsely even in models that have them.
- Educators are encouraged to highlight the value of `abstract` signatures and how they can be used to take advantage of inheritance and scope inference in addition to making the model more concise and easier to modify in the future.

RQ# 5: How often are scalars used in Alloy?

Motivation: Alloy has no construct for scalars. Sets of size one are used to represent scalars to simplify the language so that only operators over sets are needed. But this can be confusing to novice modelers because most other languages provide scalars. Syntactic sugar could be provided to allow modelers to include scalars directly in their model and have these converted to sets underneath or an analysis method could be optimized if many sets are scalars.

Approach: Alloy allows modelers to declare sets of size one using signatures that have multiplicity `one` (*e.g.,* set `B1` is a subset of `B` with size one on line 6 in Figure 4.1) or using `enum`s. **Signatures with multiplicity one** have exactly one atom in them. `Enum`s allow the instantiation of multiple signatures with multiplicity `one` concisely. Signatures

introduced using `enum` will have an ordering imposed on them. Creating multiple ordered signatures with multiplicity `one` is a multi-line procedure that can be greatly simplified using `enum`. We count signatures declared with the keyword `one`. In the post-processing stage, we ensure that declarations containing multiple signature are counted correctly and report the total number of signatures with multiplicity `one`. `Enum`s are special signature declarations, where each listed element is equivalent to a signature with multiplicity `one`.

Construct	PU	TU	D
`one sig`	1*	1	40.9% } of sigs
`enum`s	2*	0	1.5% } of sigs

Table 4.4: Use of Scalars

Results: Table 4.4 shows that signatures with multiplicity `one` account for 40.9% of all signatures declared across all the models. The typical Alloy model contains one signature declared with multiplicity `one` . We find that `enum`s account for 1.5% of all signatures. The typical use criterion for `enum`s is zero, which means that the typical Alloy model does not contain an `enum` declaration. Scalars declared using signatures with multiplicity `one` far outnumber the ones declared using `enum`s.

Findings: We find that:

- Given that `enum`s are an underutilized construct in Alloy, educators are encouraged to highlight the use of `enum`s to concisely instantiate multiple ordered signatures with multiplicity `one`.
- The abundant use of scalars in Alloy models is evident and may warrant attention from language and tool designers who should consider adding syntactic sugar that allows modelers to create scalars directly.

RQ# 6: What is the number of fields per signature in an Alloy model?

Motivation: The body of signatures in Alloy can contain fields. Alloy is often considered to have an object-oriented flavor because of the way that fields that take a particular set as their first argument are grouped with the signature of that set (similar to a method of an object). By examining the number of fields per signature, we aim to get a better understanding of the distribution of fields over signatures (*i.e.,* if modelers are grouping all fields under one signature or if they are dividing them among several signatures).

Approach: We extract and tally up the number of fields declared in the body of each signature across all models. We account for multiple fields in one declaration (*e.g.,* relations

`f3` and `f4` on line 7 in Figure 4.1). We present the percentage of signatures with and without fields. We also report the predominant and typical use criteria in addition to the percentile distribution and common range.

Construct	**Distribution**
Signatures with Fields	12.9%
Signatures without Fields	87.1%

Table 4.5: Distribution of Signatures with and without Fields

Construct	**Predominant Use** (Mode)	**Typical Use** (Median)
Fields	1	2

Table 4.6: Fields in Alloy Models

Construct	**12.5$^{\text{th}}$**	**25$^{\text{th}}$**	**50$^{\text{th}}$**	**75$^{\text{th}}$**	**87.5$^{\text{th}}$**
Fields	1	1	2	3	4

Common Range: $[1, 4]$

Table 4.7: Percentile Distribution of Field Count

Results: Table 4.5 shows that the vast majority of signatures do not have fields (87.1%) while only 12.9% of signatures have fields associated with them. Table 4.6 shows the predominant and typical use values for the number of fields per signature. Table 4.7 shows the percentile distribution and the common range. We find that in a typical Alloy model, signatures with field declarations typically have two fields. We also find that the common range for the number of fields is $[1, 4]$. We conclude that signatures commonly have between one and four fields.

Findings: We conclude that:

- Modelers are not aggregating fields under one signature but spreading them over multiple signatures.

4.3 Constraints

This section explores the use of formulas in Alloy models. We attempt to quantify formula use in Alloy models by counting the number of top-level formulas in a model. We also discuss research questions related to Alloy constructs that contain constraints. We explore the use of predicates, functions and assertions as well as fact blocks and attempt to identify trends among modelers. We also examine queries to determine how modelers are executing parameterized constraint containers (*i.e.,* predicates, functions and assertions).

RQ# 7: How many formulas does an Alloy model contain?

Motivation: Predicate, functions, assertions and facts can contain multiple formulas. Formulas express constraints and can be used in the body of predicates, functions, assertions and macros. In this research question, we count the number of top-level formulas in each model in our corpus. Top-level formulas are defined as formulas that are not part of another larger formula. We compute the top-level formula count of Alloy models to determine how they compare to programs in terms of the number of statements. We hypothesize that models with a higher top-level formula count may be more advanced.

Approach: For each model in our corpus, we extract top-level formulas and report the formula count by tallying up the number of top-level formulas. We account for predicate and function calls in the model *i.e.,* the number of formulas in a predicate or function is scaled according to the number of calls made to that predicate or function. We report the typical use criterion as well as the percentile distribution and common for the formula count.

Construct	**12.5$^{\text{th}}$**	**25$^{\text{th}}$**	**50$^{\text{th}}$**	**75$^{\text{th}}$**	**87.5$^{\text{th}}$**
Formulas	2	7	21	58	92

Common Range: $[2, 92]$

Table 4.8: Percentile Distribution of Formula Count

Results: We find that the typical use criterion for the formula count is 21 *i.e.,* a typical Alloy model contains 21 top-level formulas. Table 4.8 shows the percentile distribution of the formula count. Akin to the signature count, formula use in Alloy differs significantly from one model to another. The common range for the formula count is $[2, 92]$.

Findings: The results of this research question suggest that:

- Formulas are used abundantly in Alloy models.
- The number of top-level formulas exhibits a great deal of variation among models. Thus, Alloy models can vary significantly in terms of scale.

In Section 4.5, we regress model length against formula count to find out if there exists a correlation between these two model characteristics.

RQ# 8: How are modelers using facts?

```
1  sig A, B {} {
2    A !in B
3  }
4  sig C {
5   f: A -> lone B
6  }
7
8  sig D { from, to: A}
9  {from != to}
10
11 pred map(c, c':C, a:A, b:B) {
12  c'.f = c.f + a -> b
13 }
14
15 fun lookup(c: C, a: A): set B {
16  a.(c.f)
17 }
18
19 fact at_least_one {
20  some C
21 }
22
23 assert unique_mapping {
24  all c:C, a:A, b, b':B |
25
26  a -> b in c.f and a -> b' in c.f implies b = b'
27 }
28
29 run map for 3
30 run lookup
31 check unique_mapping
```

Figure 4.2: Alloy Constraint Holders Example (adapted from [3])

Motivation: Facts are a packaging construct that contains constraints that hold in all instances of the model. Alloy models can contain any number of facts that are grouped in separate blocks. Fact blocks can be declared using the keyword `fact` as shown on Line 19 in Figure 4.2 or coupled with a signature by placing immediately following a signature declaration (*e.g.,* the signature fact on line 9). Signature facts are constraints that apply to all elements of a signature's set *i.e.,* they are implicitly quantified over the signatures. Field references made in signature facts are implicitly dereferenced. Signature facts are similar to class invariants in an object-oriented language. Jackson's book on Alloy [37] states that the use of signature facts should be limited since the implicit quantification can lead to unexpected consequences. Figure 4.3 shows two equivalent instantiations of the signature `B`. The declaration on line 2 uses a signature fact on line 4 to formulate the constraint `f1 != f2` whereas the declaration on line 6 uses a fact block on line 7 to formulate the same constraint. Note that the constraint on line 7 has an explicit quantification over all elements of the signature `B`. The constraint in the signature fact on line 4 has the same quantification but it is implicit. The implicit quantification may not be evident to the modeler and thus the constraints in the body of a signature fact will apply to all the elements of the signatures even though that may not be the intention of the modeler. By examining the use frequency of facts, we can help educators evaluate how much to emphasize the proper use of this construct.

```
1  sig A {}
2  sig B {
3   f1, f2: A -> B
4  }  {f1 != f2}
5
6  sig B {f1, f2: A -> B}
7  fact {all x: B | x.f1 != x.f2}
```

Figure 4.3: Signature Facts

Approach: We extract and tally up the number and kinds of fact blocks in a model. We also extract from the model all signature declarations that contain a non-empty fact block. Empty signature fact blocks are ignored since they do not contain any implicitly-quantified expressions that can have unwanted results.

Results: The results of this research question are summarized in Table 4.9. We find that the typical Alloy model contains two fact blocks and one signature declaration with facts.

Fact Type	**PU** (mode)	**TU** (median)	**D**
Signature Facts	1	1	47.6% of sigs 42.9% of facts
Facts	1*	2	-

Table 4.9: Facts Query Results

We find that 47.6% of signatures have a fact block associated with them and 42.9% of all fact blocks are signature fact blocks. The percentage distribution shows that the use of signature facts is common among Alloy modelers.

Findings: We come to the following conclusion:

- Educators are encouraged to ensure that student modelers are using signature facts correctly to avoid erroneous results. Alternatively, educators may want to discourage the use of signature facts.

RQ# 9: How often do modelers declare and call predicates and functions?

Motivation: A predicate is a named constraint with zero or more arguments creating a set of formulas. Line 11 in Figure 4.2 shows an example of a predicate declaration. A function is a named expression with zero or more arguments that returns a value. Line 15 in Figure 4.2 shows an example of a function declaration. Predicates allow modelers to analyze models with constraints excluded or included. Lines 29 and 30 show an example of using a command query to run a predicate and a function respectively for model verification. Function and predicate calls may also occur in other constraint containers and are used for model description in this case. Daniel Jackson [8] states that when a function is used with a `run` command, Alloy finds an instance that makes the constraint true. In this case, the instance consists of a collection of arguments for the function, the values of signatures and fields, and the function result. Predicates and functions used with command queries are utilized for model verification whereas the ones called in formulas are used for model description. We examine the use of predicates and functions to determine their frequency in Alloy models as well as any trends that modelers adhere to when adding constraints.

Approach: We extract and tally up the number of predicate and function declarations in each model. We then extract the names of these predicates and functions from the declarations. We cross-reference these names with the ones used in command queries and formulas to identify and tally up the number of predicate and function calls in the model. We report the predominant and typical use criteria as well as the percentage distribution

of declarations and calls (in commands and formulas). We also report the percentage of models that contain predicates and functions. These values may be skewed given that modelers tend to comment out command queries frequently.

Construct	PU	TU	D	
Predicate Declarations	1	2	80.1%	= 100%
Function Declarations	1*	0	19.9%	
Predicate Calls (Commands)	1*	1	11.2%	= 100%
Predicate Calls (Formulas)	2*	1	88.8%	
Function Calls (Commands)	1*	0	0.1%	= 100%
Function Calls (Formulas)	2*	0	99.9%	

Table 4.10: Use of Predicates and Functions

Results: We find that 81.7% of all models in our corpus contain predicate declarations compared to 26.9% of models that contain function declarations. Table 4.10 shows the predominant and typical use criteria as well as the percentage distribution. The predominant and typical use values show a greater use of predicates than functions among Alloy modelers. We find that modelers declare and call predicates significantly more often than functions. A typical Alloy model contains two predicate declarations and two predicate calls but no function declarations or calls. This trend of predicate prevalence is also reflected in the percentage distribution where 80.1% of these parameterized declarations are predicates while the remaining 19.9% are functions. Similarly, predicate calls account for 77.4% of the total parameterized expression calls, with function calls accounting for the remaining 22.6%. We find that predicate use in formulas (88.8%) greatly outnumbers predicate use in commands (11.2%). Functions are almost exclusively used in formulas.

Findings: We conclude that:

- Modelers are using a small number of predicates and functions in their models.
- Predicates and functions are used more frequently for model description as opposed to model verification given that the majority of predicate and function calls occur in formulas and not in commands.
- Using functions with a `run` command is an underutilized functionality of the Alloy language. Educators are encouraged to highlight this functionality and explain to student modelers how it can be used to obtain a collection of arguments for the function, the values of signatures and fields, and the function result.

- Alternatively, language designers may want to remove the ability to `run` functions from the Alloy language given the scarcity of its use.

RQ# 10: How often are assertions used in Alloy models?

Motivation: Assertions are constraints that ensue from facts and ensure that the model cannot reach invalid states. Assertions are declared using the `assert` keyword and contain formulas that express constraints. Line 31 in Figure 4.2 shows an example of a `check` command used to find a counterexample to a constraint defined using an assertion. Assertions are a remnant of an older version of Alloy when predicates could not be used with `check` commands [9]. As of Alloy v4, assertions can be replaced with unparameterized predicates. We explore the frequency of assertions in Alloy models and identify user trends to help educators determine how much to emphasize the use of this redundant construct. Language and tool designers may want to remove assertions from the Alloy language if they are not being used frequently.

Approach: We extract and tally up the number of assertions in each model. We then extract the names of these assertions from the declarations. Next, we cross-reference these names with the ones used in `check` command queries to identify and tally up the number of assertion uses in the model. We report the predominant and typical use criteria as well as the percentage of models that have assertion declarations in them.

Construct	**Predominant Use** (Mode)	**Typical Use** (Median)
Assertion Declarations	1*	2*
Assertion Uses	1*	2*

Table 4.11: Assertion Declarations and Uses

Results: We find that 31% of Alloy model in our corpus contain assertion declarations. Table 4.11 shows the results of this research question. We present the non-zero predominant and typical use values given that the all-inclusive values were all zero. Alloy models that make use of the assertion construct typically contain two assertion declarations and two assertion uses.

Findings: We conclude that:

- Assertions are still used in Alloy models even though they are a remnant of an older version of the language.

- Educators should encourage student modelers to use unparameterized predicates instead of assertions to simplify the language.
- Language designer may want to consider removing assertions from future versions of Alloy since they are a redundant construct that is not used frequently.

RQ# 11: How often are `run` and `check` commands used?

Motivation: Alloy blurs the line between model and properties to check since they are both in the same language. Alloy provides modelers with two command queries to check the behavior of the model and its properties. A `run` command looks for satisfying instances and is applied to predicates whereas a `check` command provides a counterexample if there is an instance that does not satisfy a predicate or an assertion. One type of command can be turned into the other through negation of relevant formulas. Depending on how the model is arranged into facts and predicates, and the purpose of the model (verification vs. synthesis) one or the other type of query may be more useful. In this analysis, we also investigate the number of queries in a model overall. In answering this question, we can see whether the two ways of querying Alloy models are valuable to users.

Approach: We count the occurrences of `run` and `check` commands per file and over all models. We compute various statistics related to the number of `run` and `check` commands as well as the total number of commands in a model. Our parser ignores comments, which might contain some `run` or `check` commands.

Command	Predominant Use	Typical Use	Distribution
run	1	1	48.1%
check	1*	0	51.9%
run + check	1	1	-

Table 4.12: Use of Run and Check Queries

Results: Table 4.12 shows that modelers are using `run` and `check` commands equally as often. The typical use criterion per model for `check` commands is 1 and its `run` counterpart is zero. A typical Alloy models contains one `run` command but no `check` commands which indicates that `run` commands have a higher use frequency then `check` commands. The percentage distribution is near-equal between `run` and `check` with 48.1% of queries being `run` commands while the remaining 51.9% of queries are `check` commands.

Findings: We conclude that:

- Both command forms are used frequently and are generally useful for modelers even though they are interchangeable.

RQ# 12: How are `run` and `check` commands used in Alloy?

Motivation: `run` commands are used to find instances that satisfy a predicate or a series of constraints declared in the command itself. `check` commands are used to find counterexamples to a previously declared predicate or assertion, or they are used with a, possibly named, constraint block declared in the `check` command. `run` and `check` commands can take one of the three following forms:

- `run name` or `check name`: The command explicitly mentions the name of a previously declared predicate or assertion.
- `run {constraints}` or `check name`: The command is followed by a block containing the unnamed constraints.
- `run name {constraints}` or `check name {constraints}`: This form contains a named constraint block that follows the `run` or `check` keyword. While the constraint block in this form is technically named (possibly for documentation), it cannot be referenced by another command query or formula.

By exploring the different forms of `run` and `check` commands, we can identify trends and preferences among Alloy modelers. The results of this research question can help educators focus on teaching the command form that is most widely used by the Alloy community. Language designers can consider dropping support for command forms that are not commonly used.

Approach: We create three patterns that denote each one of the command forms and extract matching instances from the models. We report the percentage distribution over the three command forms.

Form	Percentage Distribution	
`run name`	70.6%	} = 100%
`run {constraints}`	14.0%	
`run name {constraints}`	15.4%	

Table 4.13: Use of `run` Command Forms

Form	Percentage Distribution	
`check name`	81.8%	= 100%
`check {constraints}`	9.9%	
`check name {constraints}`	8.3%	

Table 4.14: Use of `check` Command Forms

Results: Table 4.13 shows the percentage distribution of `run` commands over the three different forms. Alloy modelers use the `run` command most commonly with a named predicate (70.6%). We also find that naming the constraint block in a `run` command is the second most popular form coming in 15.3% of all `run` commands. Placing a constraint block with the `run` command is the least popular form coming in at 14%. Table 4.14 shows the percentage distribution `check` commands over the three forms. The vast majority of `check` commands are used with previously declared assertions (81.8%). The two other forms are quite sparse in Alloy models with the unnamed blocks coming in at 9.9% and unnamed blocks coming in at 8.3%.

Findings: The findings of this research are listed below:

- Language designers may want to consider removing the ability to name constraint blocks in `run` and `check` commands since it is not widely used and it does not offer any advantages in terms of reusing the constraints.

4.4 Corner Cases

Similar to every language, the Alloy language has some features that are generally considered either more complicated to use, less likely to be useful, or problematic, based on the literature. In the following three research questions, we explore whether these corner cases are actually rarely used or not. We explore the use of five corner-case features of the Alloy language: subsets of a union of signatures, bit shifting operators, set constants, integer use and macros.

RQ# 13: How often are modelers creating subsets of union of signatures?

Motivation: Signatures in Alloy can be declared as a **subset of a union of signatures** meaning the elements belong to any one of the sets referenced in the union. The ambiguity

surrounding the parent of the declared signature can lead to unexpected results. Union supersets are considered uncommon in Alloy, so we set out to count the occurrences of this corner case.

Approach: We identify signature declarations where the superset is the union of signatures and report the total number of these signatures.

Results: There are only 23 instances of union supersets across all models, which aligns with our initial assumption concerning the uncommonness of this practice.

Findings: We find that:

- Optimizations developers can safely omit support for union supersets when developing their optimizations because they are used scarcely in Alloy models.

RQ# 14: How common is the use of bit shifting operators in Alloy models?

Motivation: Alloy includes three bit shifting operators: the **left-shift** `<<`, the **sign-extended right-shift** `>>` and the **zero-extended right-shift** `>>>`. Given Alloy's abstract nature, bit operations are quite uncommon. By examining the use of bit shifting operators we help language designers and educators determine how much to focus on the development and teaching of these constructs.

Approach: We count the use of bit shifting operators by extracting any application of these operators.

Results: We only identified nine occurrences of the bit shifting operators in our repository of models, which confirms the scarceness of bit operations in Alloy. These occurrences were found in three Alloy models only.

Findings: Based on the results of this research question, we recommend that:

- Language designers should consider dropping support for bit shifting operators in future versions Alloy so that solving engines do not have to support these operations.

RQ# 15: How common is the use of set constants in Alloy models?

Motivation: Alloy provides modelers with three constants: `none`, `univ`, and `iden`. The unary sets `none` and `univ` are the sets containing no elements and every element respectively. The binary identity relation `iden` contains a tuple relating every element (in the universe) to itself. To ensure the generality of some predicate declarations, the empty set

can be passed as an argument. For instance, consider the following predicate declaration `pred p(a: A, b: B)` where p takes one argument of the kind `A` and another one of kind `B`. The following call to `p` using the empty set `p(a, none)` is a valid predicate call in Alloy. The identity relation is required for some constraints expressed in the relational calculus style (*e.g.,* `no ^r & iden` indicates that the relation r is acyclic). Jackson's book on Alloy [37] states that these constants are rarely used aside from the aforementioned cases.

Approach: We extract all references to set constants and tally the occurrences of each constant separately. We also count the number of models that use constants.

Constant	PU	TU	D
`none`	1*	4*	7.3%
`univ`	2*	3*	4.5% } of all models
`iden`	1*	1*	4.9%

Table 4.15: Use of Constants by Model

Constant	Percentage Distribution
`none`	40.0%
`univ`	52.3% } = 100%
`iden`	7.7%

Table 4.16: Constant Use

Results: Table 4.15 shows the data summary criteria computed for this research question. We note that the non-zero typical use criteria for `none` and `univ` (*i.e.,* only for models that have these constants) are fairly high, which leads us to believe that while constants are generally uncommon, they are used profusely in models that require them. Table 4.16 shows that `univ` is the most-used constant with 52.3% of the uses of constants being `univ` followed by `none` and `iden`.

Findings: Based on the results of this research question, we recommend that:

- The frequency of set constant use aligns with our initial expectations. Set constant are used sparsely overall as presumed in Daniel Jackson's book on Alloy [37].

RQ# 16: How common is the use of macros in Alloy models?

Motivation: Macros (also known as untyped macros or `let` statements) in the Alloy language are similar to predicates and functions. However, macros are expanded before runtime and can be used as part of signature fields. Parameters in macros are not required to have a type. Consequently, macro parameterization is more flexible and allows the use of arbitrary signatures, boolean constraints, sets, relations and even calls to predicates, functions and other macros. Macros are defined using the `let` keyword at the top level of a model. Macro declarations can take one of three forms that differ in their use of `=` and `{ }`. Figure 4.4 shows all three forms of macro declarations. Line 11 shows the macro `m2` being used to create a field `f`.

```
 1  sig A {}
 2
 3  sig B {}
 4
 5  let m1[x] = { x one -> A }
 6
 7  let m2[x] = x -> A
 8
 9  let m3 [a,b,c] { a[b,c] }
10
11  sig C {f: m2[B] }
```

Figure 4.4: Macros in Alloy

Approach: We extract and tally up the number of macro blocks in each model. We also count and report the number of models that make use of macros.

Results: We find 166 macro declarations across our entire repository of models. We also find that 2.4% of all models contain macro declarations (52 models).

Findings: We conclude that:

- It is evident that macros are not as prevalent as predicates and functions in Alloy models.
- Given their untyped nature, macros are significantly more flexible than predicates and functions and can be used in ways that other constraint holders cannot. Educators are encouraged to highlight macros and promote their use when teaching student modelers.

RQ# 17: Are modelers creating fields declared using set union and set difference?

Motivation: Alloy allows modelers to declare fields using set union (*e.g.,* `f: A + B`) and set difference (*e.g.,* `f: A - B`). These uncommon practices arise from the simplicity of the Alloy language. Fields declared using set union and set difference can be problematic for external tools and engines that implement optimizations because they are special cases. By examining the frequency of these fields, we can provide developers working on optimizations with useful insight that can help them determine whether or not to address these cases.

Approach: Fields declared using set union and set difference are extracted from the body of signature declarations. We tally up the number of these fields and report the percentage distribution over all fields and models.

Results: We find that fields declared using set union and set difference constitute 1.96% of all fields with set union being the overwhelming majority of these fields (1.9%) and set difference accounting for 0.06% of all fields only. We identify a similar trend in the model distribution. Models that contain fields declared using set union and set difference account for 1.4% of all models (models with fields declared using set union constitute 1.2% of these models with the remaining 0.3% being models with fields declared using set difference).

Findings: We come to the conclusion that:

- Fields declared with set union and set difference are quite rare in Alloy.
- Developers can safely omit support for fields declared using set union and set difference in their optimizations without significantly compromising the usefulness of the tools.
- Given the scarcity of fields declared using set union and set difference, language designers may want to remove support for these expressions in fields to simplify the language.

4.5 Correlating Model Characteristics

In this section, we explore the correlations that exist between various model characteristics discussed in previous research questions. The main statistical tool used to perform this

analysis is the linear regression model discussed in Section 3.3 of Chapter 3. We present a series of research questions that correlate certain Alloy model characteristics using linear regression. We examine the correlations that exist between model length, the number of formulas and the number of sets to determine if longer Alloy models have more sets or more formulas. We also correlate the number of fields with the number of top-level signatures and the number of subsignature extensions to determine which signature kind count gives us a better insight into the size of the state space.

RQ# 18: Is model length correlated with the number of sets or the number of formulas?

Motivation: We have previously established that Alloy models tend be relatively short. Nevertheless, there is some variation in the length of Alloy models. In this research question, we attempt to identify if model length is affected by the number of sets or the number of formulas. Our findings can help language and tool designers get a better understanding of the structure of longer models.

Approach: For each model in our corpus, we compute and record the length (excluding blank lines and comments), the set count and the top-level formula count. The set count includes all signatures and fields declared in the model. The top-level formula count is limited to top-level formulas in the model *i.e.,* formulas that are not part of any other larger formula. The number of top-level formulas in the body of predicates and functions is scaled according to the number of calls made to the constraint holder. We limit these measurements to models that have non-zero set and top-level formula counts to ensure that the logarithmic transformation can be applied. This procedure limits our corpus size to 1,935 models out of 2,138 models. We perform linear regression to produce the best fit line for Model Length vs. Number of Sets and Model Length vs. Number of Formulas. We also produce all four residual plots for each linear regression model.

Results: Figure 4.5 shows the Model Length vs. the Number of Sets in the model. The r value of 0.8 suggests a high positive correlation between the set count and the model length. However, with a conservative goodness of fit ($R^2 = 0.63$), this correlation only explains 63% of the variation in the data set. Figure 4.6 shows the residual plots of the Model Length vs. Number of Sets linear regression model. These plots suggest that the linear model is a good fit given that the Residual vs. Fitted and the Scale-Location plots show an almost horizontal line with data points distributed on both sides. The Normal Q-Q plot is also indicative of a good fit since the point form an approximately straight line with minor deviations on both ends. We also find that no data points fall outside of Crook's distance line in the Residuals vs. Leverage graph (the lines themselves are not visible on the graph due to the large distance between them and the data points). The

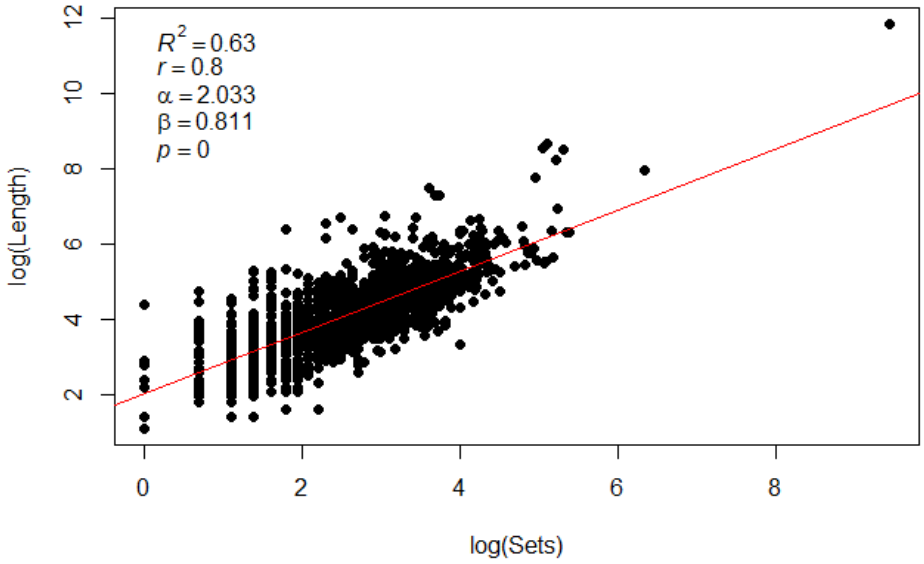

Figure 4.5: Model Length vs. Number of Sets in *log* scale

residual plots indicate that the fit cannot be significantly improved beyond $R^2 = 0.63$. Since the coefficient of determination R^2 is relatively conservative, we conclude that the high correlation between set count and model length is not applicable to a large portion (37%) of the subset of models used in this research question. Therefore, set count is not good predictor for model length. Nevertheless, for the 63% of data points to which this correlation is applicable, the length of a model can be approximated using the following equation derived from equation 3.1:

$$Length = e^{2.036} Sets^{0.8094} \tag{4.1}$$

Given that $\beta = 0.8094 < 1$, we find that as the number of sets grows, the model length grows at an exponentially slower pace. This can be demonstrated by plugging values into equation 4.1. For example, if a model has 2 sets, then equation 4.1 predicts that it will be approximately 13 lines in length (excluding blank lines and comments). A model with 4 sets will have 24 lines and a model with 8 sets will have 41 lines. As the number of sets grows by a factor of 2, the number of lines grows as well but by a smaller factor.

Figure 4.7 shows the Model Length vs. the Number of formulas in the model. We

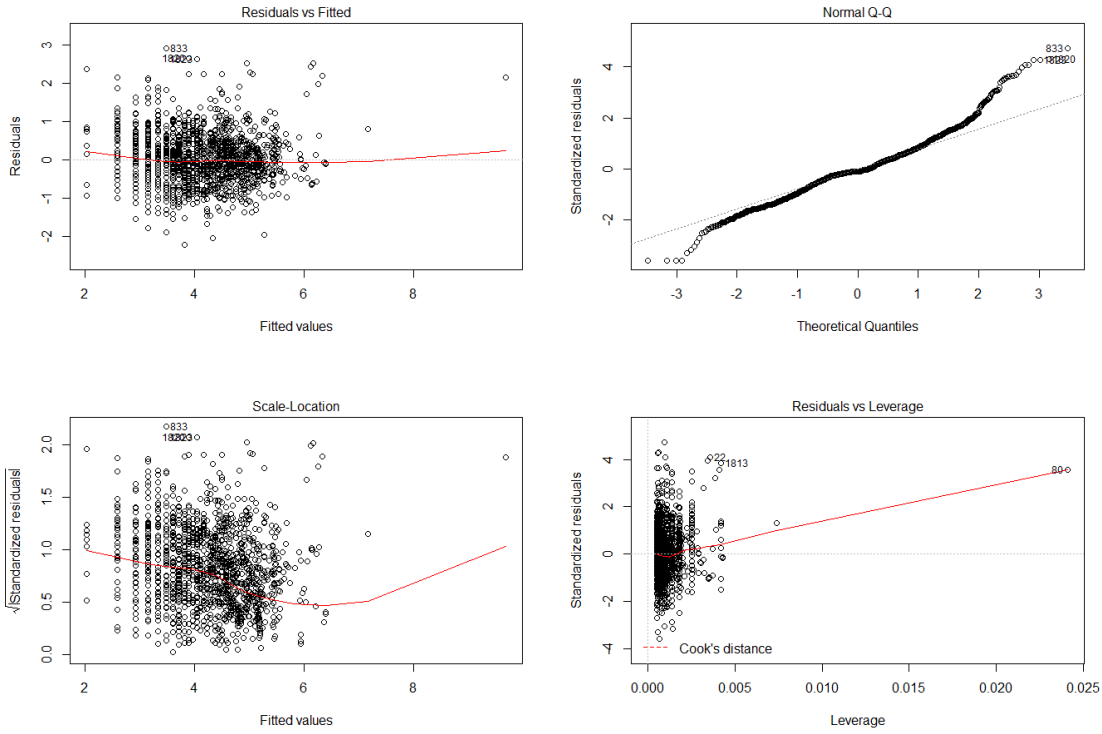

Figure 4.6: Residuals of Model Length $\sim$ Number of Sets

find that $r = 0.92$ which indicates a strong positive correlation between formula count and model length. Similar to Figure 4.6, all the residual plots conform to the previously established characteristics of a good linear regression model fit. With 84% linear fitness (in log space) and $\beta = 0.7126 < 1$, it appears that as the number of formulas grows, the model length also grows but at exponentially slower pace. Specifically, using equation 3.1, we find that:

$$Length = e^{2.056} Formulas^{0.7126} \tag{4.2}$$

For instance, if a model has 10 top-level formulas, then we can predict it will be approximately 40 lines in length (excluding blank lines and comments). A model with 100 top-level formulas has 208 lines whereas a model with 1000 top-level formulas has 1073 lines. Notice that when the formula count grows by a factor of 10, the length grows by a factor significantly smaller than 10 (as a result of $\beta < 1$). Figure 4.8 shows the residual plots of Model Length vs. Number of Formulas.

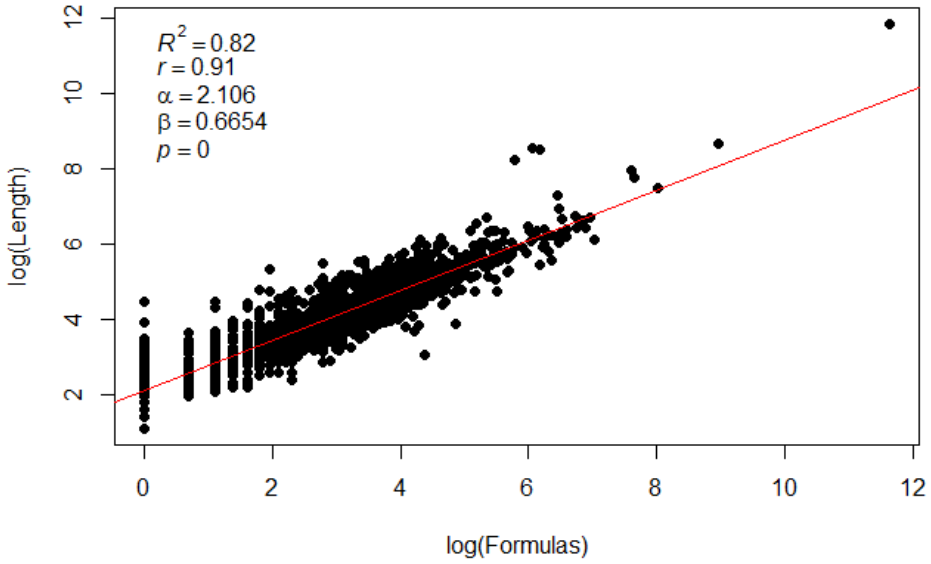

Figure 4.7: Model Length vs. Number of Formulas in *log* scale

Findings: Based on the produced linear regression models, we conclude that:

- The number of sets is not necessarily a good predictor for the length of an Alloy model.
- There exists a strong positive correlation between Alloy model length and the top-level formula count that applies to the vast majority of models in our corpus.
- Longer Alloy models will probably have more formulas but not necessarily more sets.

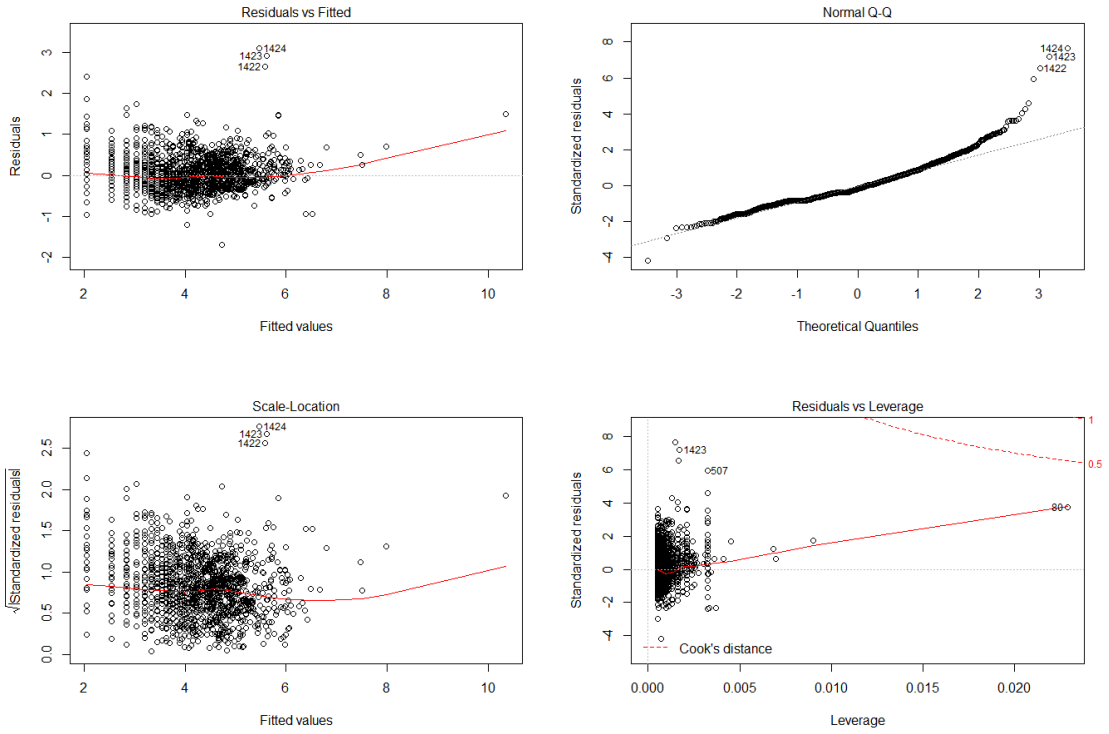

Figure 4.8: Residuals of Model Length ∼ Number of Formulas

RQ# 19: Which signature kind is the most strongly correlated with the number of fields?

Motivation: In prior research questions, we examined signature distribution by level (top-level, extensions and subsets) as well as the number of fields per signature. We found that top-level signatures and subsignature extensions are the two most frequently used signature kinds in Alloy models (top-level signature account for 16.5% of all signatures whereas subsignature extensions constitute 82.3% of all signatures). In this research question, we attempt to find which signature level between these two is most strongly correlated with the field count. The size of the state space of an Alloy models increases as more fields are declared under signatures. Thus, the signature kind with the higher correlation with the field count is a better indicator of the size of the state space.

Approach: From each model in our corpus, extract and tally up top-level signatures and

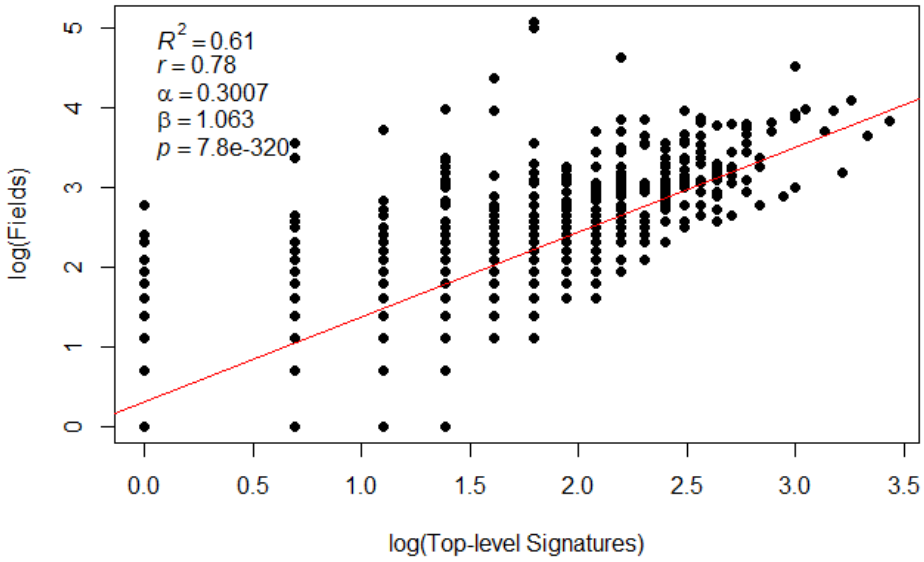

Figure 4.9: Number of Top-level Signatures vs. Number of Fields in *log* scale

subsignature extensions. We also report the total number of fields in each model. We only consider models where both the number of top-level signatures or subsignature extensions and the number of fields are not zero to ensure that the logarithmic transformation can be applied. Thus, we are limited to 1,553 files for the Number of Top-level Signatures vs. Number of Fields regression model and 1,113 models for the Number of Subsignature Extensions vs. Number of Field regression model. We produce and plot the best fit lines for both models in addition to the residual plots.

Results: Figure 4.9 shows the field count in a model plotted against the number of top-level signatures. The computed value of the coefficient of correlation r is 0.78 and thus indicates a high positive correlation between the field count and the number of top-level signatures. Nevertheless, the fit is only applicable to 61% of the data points given that value of R^2 is 0.61. The residual plots shown in Figure 4.10 adhere to the characteristics of a good linear regression model fit. The Residuals vs. Fitted plot and the Scale-Location plot show near-horizontal lines with data points on both sides. The Normal-QQ plot is a quasi-straight line with some deviations on both ends. We also note that no data points in the Residuals vs. Leverage plot exist outside of Crook's distance lines. The following equation can be

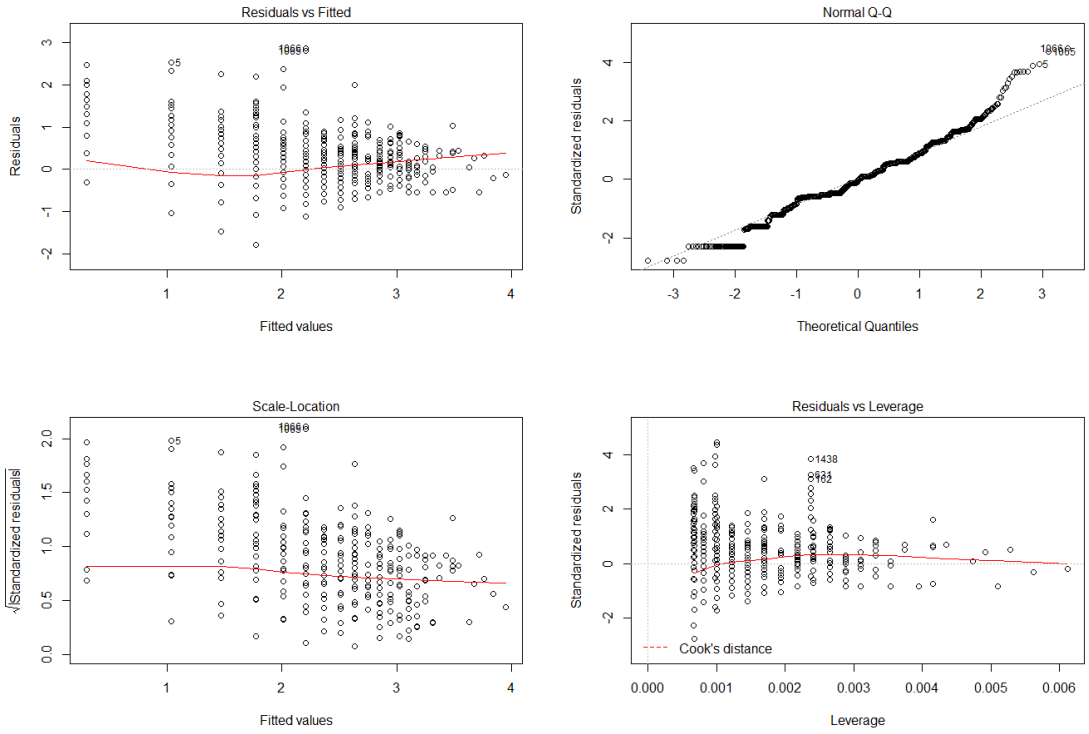

Figure 4.10: Residuals of Number of Top-level Signatures ~ Number of Fields

used to derived the number of fields given the number of top-level signatures in a model:

$$Fields = e^{0.3007}\, Top\text{-}level\ Signatures^{1.063} \tag{4.3}$$

For instance, a model with 2 top-level signatures has approximately 3 fields according to equation 4.3. A model with 4 top-level signatures has 6 fields. Since $\beta = 1.063 \approx 1$, the function is almost linear. The number of fields grows approximately by a factor of $e^{0.3007} \approx 1.3508$ as the number of top-level signatures grows.

Figure 4.11 shows the best fit line that results from regressing the number of subsignature extensions against the number of fields. The r value of 0.42 indicates a very low correlation between the field count and the number of subsignature extensions. The coefficient of determination R^2 is also very low at 0.18 which implies that this low correlation is only applicable to 18% of the data points. It is evident that there does not exist a statistically significant correlation between the number of fields and the number of subsig-

nature extensions. The fit cannot be significantly improved given that the residual plots in Figure 4.11 adhere to the characteristics of a good regression model fit.

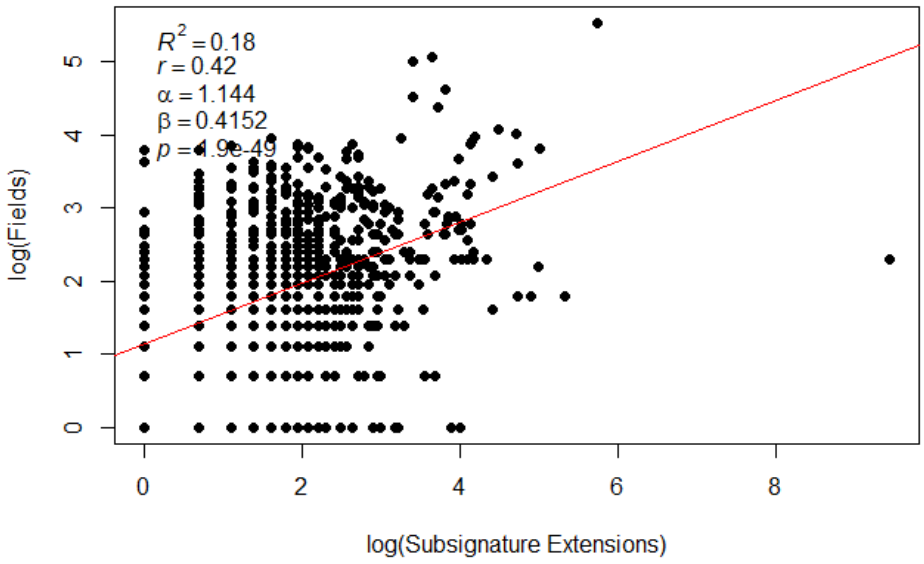

Figure 4.11: Number of Subsignature Extensions vs. Number of Fields in *log* scale

Findings: Based on the linear regression models presented in this research question, we conclude that:

- Top-level signatures are more strongly correlated with the field count of an Alloy model compared to subsignature extensions.
- The number of top-level signatures is an adequate predictor of the number of fields in a model.
- Most fields in Alloy models are declared under top-level signatures which suggests that the number of top-level signatures is a good measure of the state space.

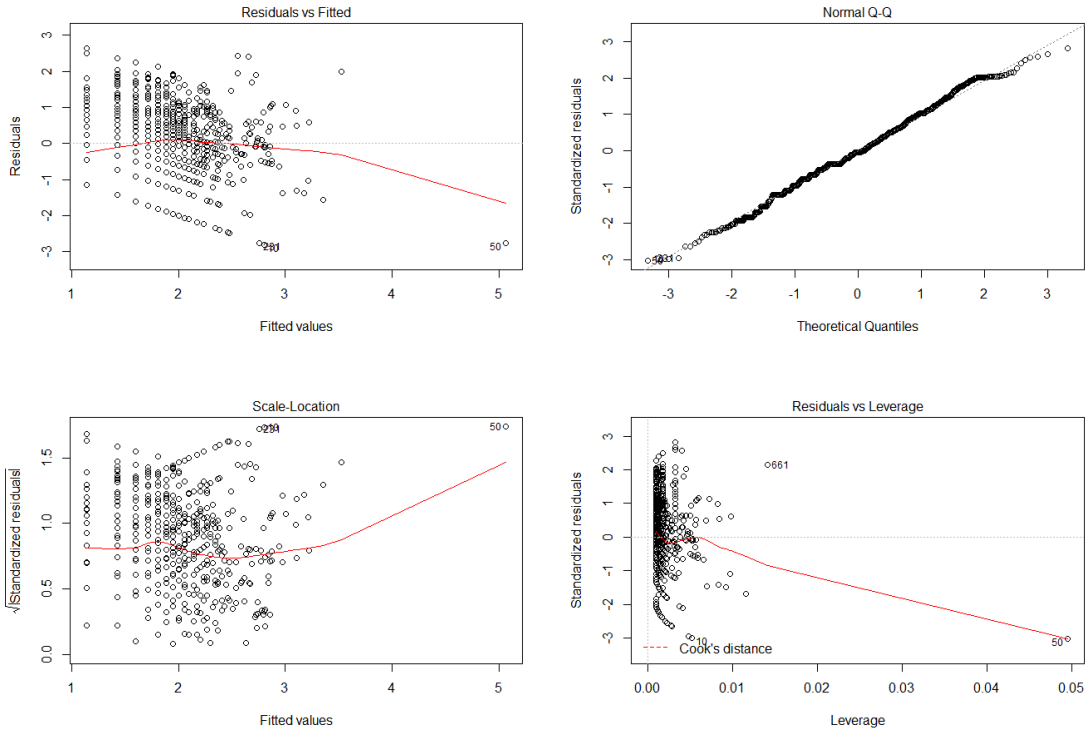

Figure 4.12: Residuals of Number of Subsignature Extensions ∼ Number of Fields

4.6 Summary

In this chapter, we explore the characteristics of Alloy models through "surface-level" research questions that examine the use of the language's constructs. Section 4.1 explores the size of Alloy models in terms of number of lines. In Section 4.2, we investigate the frequency and kinds of signatures in Alloy models. Section 4.3 discusses formulas as well as the different constraint containers used in Alloy. Section 4.4 examines uncommon or discouraged practices in Alloy known as the corner cases of the language. Lastly, Section 4.5 correlates certain model characteristics using linear regression. Our linear regression analysis indicates that the length of an Alloy model is significantly more impacted by the number of formulas than by the number of sets (*i.e.,* signatures and fields). We also found that the number of top-level signatures is a good indicator of the number of fields

in a model. We present a summary of our findings split across three categories: Findings for Language Designers, Findings for Educators and Findings for Optimization Developers.

Findings for Language Designers:

- Types are commonly used in other languages to partition a universe of atoms. Types also allow for type checking. Language designers are encourage to explore type checking mechanisms for the Alloy language to provide faster feedback to users.
- The abundant use of scalars in Alloy models is evident and may warrant attention from language and tool designers who should consider adding syntactic sugar that allows modelers to create scalars directly.
- Language designers may want to remove the ability to `run` functions from the Alloy language given the scarcity of its use.
- Language designer may want to consider removing assertions from future versions of Alloy since they are a redundant construct that is not used frequently.
- Language designers may want to consider removing the ability to name constraint blocks in `run` and `check` commands since it is not widely used and it does not offer any advantages in terms of reusing the constraints.
- Language designers should consider dropping support for bit shifting operators in future versions Alloy so that solving engines do not have to support these operations.
- Given the scarcity of fields declared using set union and set difference, language designers may want to remove support for these operations in field declarations to simplify the language.

Findings for Educators:

- Educators are encouraged to highlight the value of `abstract` signatures and how they can be used to take advantage of inheritance and scope inference in addition to creating models that are more concise and easier to evolve.
- Given that `enum`s are an underutilized construct in Alloy, educators are encouraged to highlight the use of `enum`s to concisely instantiate multiple ordered signatures with multiplicity `one`.

- Educators are encouraged to ensure that student modelers are using signature facts correctly to avoid erroneous results. Alternatively, educators may want to discourage the use of signature facts.
- Using functions with a `run` command is an underutilized functionality of the Alloy language. Educators are encouraged to highlight this functionality and explain to student modelers how it can be used to obtain a collection of arguments for the function, the values of signatures and fields, and the function result.
- Educators should encourage student modelers to use unparameterized predicates instead of assertions to simplify the language.
- Given their untyped nature, macros are significantly more flexible than predicates and functions and can be used in ways that other constraint holders cannot. Educators are encouraged to highlight macros and promote their use when teaching student modelers.

Findings for Optimization Developers:

- Optimizations developers can safely omit support for union supersets when developing their optimizations.
- Developers can safely omit support for fields declared using set union and set difference in their optimizations without significantly compromising the usefulness of the tools.

Chapter 5

Patterns of Use within Models

In this chapter, we investigate research questions that have to do with how modelers use the language's constructs and how they express descriptions in Alloy. As a flexible language, there are often multiple ways to describe the same concept in the Alloy language. These research questions are far more intricate and require multiple complex patterns and several traversals through the parse tree. These "deeper-level" questions require considerable post-processing and utilize several external data structures. The post-processing step in these questions aims to refine the data, store values in intermediary data structures and perform calculations per file. By learning about the styles and practices most commonly used in Alloy models, language and tool designers can improve and adapt Alloy based on the modelers' needs and preferences. Educators can ensure students are using the appropriate constructs for each task and encourage them to use these constructs and libraries to their full extent. Furthermore, we can learn if Alloy modelers are "purists" in consistently using one style throughout a model.

5.1 Modules

In this section, we explore the use of Alloy library modules and user-created modules. We identify the frequency of use of each library module and attempt to determine if Alloy modelers are creating models that span multiple files.

RQ# 20: How commonplace is the use of modules in Alloy?

Motivation: Alloy models can be split over multiple files. Prior to Alloy version 4, a file that is to be included in another file contained a `module` header but starting with Alloy version

4, any model can be imported by another model even if it does not contain a `module` header. We explore the use of `open` statements to include user-created modules and library modules to get a better understanding of the file structure of Alloy models. The Alloy Analyzer provides modelers with eleven library (util) modules for common operations. It is possible to describe the needed sets/relations/formulas from the library modules directly in a custom manner in one's model. However, reusable components are generally considered a good practice. Does the Alloy community buy into reusability?

Approach: We count the uses of `open` commands in a model. If a library is paramaterized, each instance of the use of a library is counted. Implicit imports of the ordering module via the use of `enum` declarations are also counted. We begin by extracting all the `open` commands in the model and classifying modules as user-created (*i.e.,* they do not contain the string `util`) and standard library modules (*i.e.,* they contain the string `util`). We then tally up the occurrences of user-created modules and each one of the eleven utility modules. Integers in Alloy (*i.e.,* the `Int` set or numeric constants) can be used without explicitly importing the integer module. For instance, field declarations whose range is `Int` count as integer uses in Alloy. Similarly, numeric constants (both positive and negative) are counted as uses of the integer module. Using the arithmetic functions of the integer module is the main reason modelers import the integer module. Therefore, we separate the use of integers into uses by importing the library and uses without importing the library.

Library module	**Distribution over `open` Statements**	**Distribution over Models**
user-created	41.8%	28.3%
ordering	35.1%	23.1%
integer	12.2%	11.3%
integer w/o import	-	32.9%
boolean	6.9%	6.3%
relation	2.2%	2.1%
graph	0.6%	0.5%
ternary	0.5%	0.4%
seq	0.3%	0.2%
naturals	0.2%	0.2%
time	0.2%	0.1%
seqrel	0.0%	0.0%
sequence	0.0%	0.0%

Table 5.1: Usage of User-Created and Library Modules

Results: Table 5.1 shows the distribution of user-created and library modules over `open` statements and over files. User-created modules account for 41.8% of all `open` statements. We also find that 29.3% of all models in our corpus contain `open` statements that import user-created modules. The ordering module, which constrains a set to be a linear order, is the most frequently opened library by a large margin being used in 35.1% of open statements and 23.1% of all models. The integer and boolean modules come in second and third place respectively. We found no uses of the `seqrel` and `sequence` modules in our corpus of models. Integers are used abundantly in ways that do not involve importing the integer module. In total, 44.2% of all models use integers but only 11.3% of these models import the integer library module explicitly, whereas the remaining 32.9% do not.

Findings: The results of this research question suggest that:

- Given the prevalence of `open` statements used to include user-created modules, Alloy language and tool designers may want to consider developing model management IDEs.
- Some library modules are rarely used and hence language designers may want to drop them in future versions of Alloy. Alternatively, educators can better highlight the value of these library modules.
- Developers are encouraged to create optimizations for the ordering module and for integers in Alloy because they are used frequently.
- Modelers are making use of integers frequently but are not taking advantage of the arithmetic operations (addition, subtraction, multiplication, division, and remainder) offered by the integer module. We further explore the use of integers in Section 5.2.

5.2 Integers

Integer use is often discouraged in Alloy. According to Jackson's book on Alloy [37], most problems with integer values do not require integers to be modeled and would benefit from more abstract descriptions and constraints. In this section, we examine the use of integers in Alloy to help educators determine how much to emphasize alternative modeling techniques that can replace integers.

RQ# 21: How are integers used in Alloy models?

Motivation: Integers in Alloy fall into two categories: integer constants and integers used in fields. Integer constants are often used to express constraints (*e.g.,* set cardinality constraints, division of sets into subgroups based on number of elements, *etc.*). Integers can appear in field declarations. Alloy provides rudimentary support for arithmetic operations through the integer module.

Approach: We extract integer constants used in expressions as well as uses of the integer set (`Int`) in field declarations. We tally up and report the number of uses that fall under each category.

Use	PU	TU	D
Constants	1*	5*	97.9%
Fields	1*	2*	2.1%

Table 5.2: Use of Integers

Results: Table 5.2 shows the data summary criteria for integer use in Alloy. We find that constants account for the overwhelming majority of integer uses (97.9%), whereas integers in field declarations constitute only 2.1% of integer uses.

Findings: We recommend that:

- Integers in Alloy take more time in analysis. Language designers should add built-in identifiers that model numeric constants (*e.g.,* `One`, `Two`, *etc.*) to satisfy the majority of integer uses without actually using integers for analysis.

RQ# 22: How often are integers used only as a linear order?

Motivation: Modelers often turn to integers as a way of modeling a linear order. The use of integers in Alloy generally takes more time in analysis because they are represented as bit vectors with bit vector operations. With this question, we are determining whether the models really need integers or whether a linear order (via the ordering module) is sufficient. It is standard advice in the Alloy community to try to avoid using integers.

Approach: Integer fields used exclusively with relational operators (meaning no arithmetic operations) can be turned into an ordering. However, this substitution is not possible for integer fields used with an arithmetic operator from the integer module. Starting with

Alloy version 4, unambiguous applications of the addition (`add`) and subtraction (`sub`) functions can be replaced with their corresponding mathematical operator. For instance, `a.add[b]` and `a.sub[b]` can be substituted for `a+b` and `a-b` respectively. Nevertheless, both forms are considered applications of an arithmetic operator from the integer module. We identify all integer values as well as all expressions containing relational and arithmetic operators. We then partition the set of integer values depending on the operator they are coupled with.

Results: We find that 55.3% of all integers used in fields do not need to be integers and could be a set with a linear ordering. This optimization is not possible for 22.2% of all integer fields. The remaining 22.5% of integer fields were not used in the model post declaration. We conclude that Alloy modelers are often not taking advantage of the ordering module.

Findings: The findings of this research question are as follows:

- Educators are encouraged to promote the use of the ordering module.
- Language designers could modify the Alloy Analyzer to allow it to warn users about integer constants that are only used as a linear order.
- Developers of Alloy optimizations are encouraged to convert these integer uses to an application of the ordering module before analysis.

RQ# 23: Are constant integers used to specify the size of sets in Alloy?

Motivation: The set cardinality operator (`#`) in Alloy allows modelers to specify the size of a set *e.g.,* `#A = 2` constrains the size of the signature `A` to be two. The set cardinality operator can also be applied to fields and expressions that denote sets. Uses of the set cardinality operator that serve to specify the size of a signature can be replaced by command query scopes or multiplicity keywords in the signature declarations. Setting set sizes using the cardinality operator often results in slower solving times [34]. Jackson's book on Alloy also discourages the use of the set cardinality operator with integers to designate the size of a signature set [37]. We explore the use of set cardinality with integers to help educators determine how much to emphasize the importance of avoiding this practice. Language designers can also use our findings to determine if a warning should be added to the Alloy Analyzer when a modeler attempts to set the size of a signature using the set cardinality operator.

Approach: We extract from the model all expressions where the set cardinality operator is used with a relational binary operator and an integer constant. We split these expressions into two categories:

- Expressions that can be turned into scope limitations
- Expressions that can be turned into formulas

Expressions that can be turned into scopes consist of the set cardinality operator applied to a signature set with the equality operator (`=`) or the less than or equal operators (`=<` or `<=`) (*i.e.,* `# <sig> = <num>`, `# <sig> =< <num>` or `# <sig> <= <num>`).

Expressions that can be turned into formulas consist of the set cardinality operator applied to a signature set with the operators `</>/>=` or a formula with any relational operator (*i.e.,* `# <sig> </>/>= <num>` or `# <formula> </>/>=/ =/<=/=> <num>`). For instance, if `set_member` is a predicate used to denote set membership, then the expression `# set_member = 2` can be replaced by the following formula:

```
some x: A | some y: A | set_member[x] and set_member[y] and x
   != y and all z: A | set_member[z] implies z = x or z = y
```

as shown in [41].

Next, we classify these applications accordingly and extract the integer constants to produce a percentile distribution and a common range for these numeric values. We present the percentage of set cardinality uses pertaining to each category.

Expression	**Percentage Distribution**	
Set Cardinality without Relational Operators	72.5%	
`# <sig> =/=</<= <num>`	12.9%	} = 27.5%
`# <sig> </>/>= <num>`	3.6%	
`# <formula> </>/>=/=/<=/=> <num>`	11.0%	

Table 5.3: Uses of Set Cardinality

Construct	**12.5th**	**25th**	**50th**	**75th**	**87.5th**
Integers	1	1	2	2	4

Common Range: $[1, 4]$

Table 5.4: Percentile Distribution of Integer Constants in Set Cardinality Uses

Results: The results of this research question are presented in Tables 5.3 and 5.4. 72.5% of all set cardinality operators are either used with numeric operators or with a combination

of numeric and relational operators and thus cannot be converted to scope limitations or formulas. We find that set cardinality uses that specify the size of a set using integers account for 27.5% of all set cardinality uses. Almost half of these expressions can be turned into command queries with scopes (12.9% of all set cardinality applications). Expressions that can be turned into formulas account for 14.6% of all set cardinality applications. Table 5.4 shows the percentile distribution of the integer constant values in set cardinality uses. We find that 75% of all integer constants fall in the range $[1, 4]$ which shows that the integer values used with the set cardinality operator are quite low. Our results suggest that the use of set cardinality to specify the size of a signature set instead of using multiplicity keywords or command queries is considerable, but not abundant.

Findings: Based on the results of this research question, we find that:

- Language designers may want to include a warning that discourages using integers to set the size of sets.
- We encourage educators to highlight the proper use of multiplicity keywords and setting scopes in command queries.
- Developers of Alloy optimizations may want to internally transform these uses of set cardinality into signature declarations with a multiplicity keyword or into command queries.

The three research questions in Section 5.2 examined the use of integers in Alloy models and found that most integer uses can be replaced with alternative constructs that conform to the guidelines provided in the language's literature. Hence, educators are encouraged to help student modelers explore alternative ways of modeling problems with numerical components that do not require integer constants.

5.3 Sets

Signatures and fields are all sets in Alloy. These sets can be connected together through constraints and hierarchy structures. This section explores the use of sets in Alloy as well as the hierarchies that ensue from the relationships between them. We devise metrics that aim to quantify set bindings and hierarchies.

RQ# 24: How deep/wide are the set hierarchy graphs in Alloy models?

Motivation: Subsignature extensions in Alloy allow modelers to introduce new subsets of the parent signature declared using the keyword `extends`. The parent-children relationships between subsignature extensions create a set hierarchy that can be modeled as a graph. For instance, Figure 5.2 shows the set hierarchy graph corresponding to the signature declarations shown in Figure 5.1. Signatures `A`, `B` and `C` are top-level signatures. `A1` and `A2` are subsignature extensions of A declared on line 4. `A3` extends `A1` as shown on line 3. Signature `B` and its extensions `B1` and `B2` are declared using `enum` on line 4. By building and exploring the characteristics of set extension hierarchy graphs (depth and width), we can get a better understanding of how modelers create sets and extend them. Set hierarchy graphs also give us an insight into the intricacy of Alloy models. Deep extension hierarchy graphs may indicate complex models.

```
1  abstract sig A {}
2  sig A1, A2 extends A {}
3  sig A3 extends A1 {}
4  enum B {B1, B2}
5  sig C {}
```

Figure 5.1: Signature Declarations

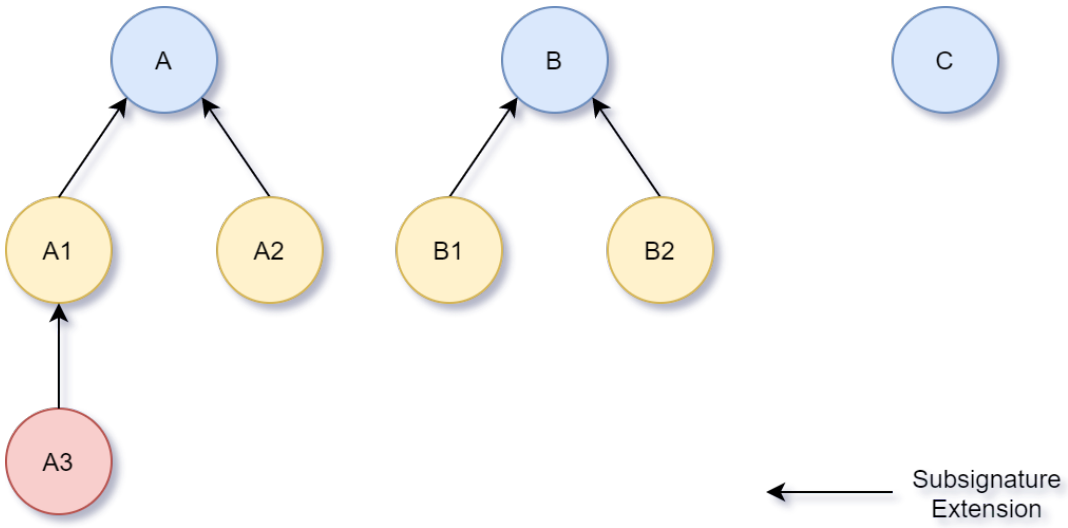

Figure 5.2: Extension Hierarchy Graph

Additionally, subset signatures in Alloy allow modelers to introduce inclusive subsets *i.e.,* an element belonging to the parent signature may or may not also belong to the subset signature. The superset of a subset signature can be a union of signatures. In this case, elements in the subset signature may belong to either one of the parent signatures in the union. Figure 5.3 introduces two additional subset signatures declared using the keyword `in` compared to Figure 5.1. `D1` declared on line 7 is a subset signature of the union `A2 + B1` and `D2` declared on line 8 is a subset of `D1`. Figure 5.4 expands the hierarchy graph in Figure 5.2 by introducing two subset nodes `D1` and `D2`. We use dotted edges to denote a subset relation between two signature nodes. Note that the node `D1` appears under `A2` and `B1` since its elements belong to either one of the parent signatures in the union. Subset signatures cannot have subsignature extensions and consequently any child node of a subset signature must also be a subset signature itself.

```
1  abstract sig A {}
2  sig A1, A2 extends A {}
3  sig A3 extends A1 {}
4  enum B {B1, B2}
5  sig C {}
6
7  sig D1 in A2 + B1 {}
8  sig D2 in D1 {}
```

Figure 5.3: Signature Declarations with Subset Signatures

Approach: We build the **extension hierarchy graph** iteratively over multiple steps. We start by extracting all signature extension declarations from the model. We then iterate over the list of signature extension declarations and extract the name of the top-level parent signature as well as the names of the extensions from each declaration. We build the hierarchy graph by adding nodes corresponding to each parent signature and its extensions and creating edges between them. We repeat this process for `enum` declarations. Finally, we add any remaining top-level signatures that are not already in the hierarchy graph. These remaining top-level signatures correspond to signatures that are not extended after being declared (*e.g.,* signature C in Figure 5.1). Once the hierarchy graph is built, we compute its depth and width. The depth of an extension hierarchy graph corresponds to the number of edges on the longest downward path between a top-level parent node and a leaf node. The depth of the hierarchy graph shown in Figure 5.2 is 2 (the longest downward path is the one between `A` and `A3`). The width of an extension hierarchy graph corresponds to the number of extension leaf nodes (*i.e.,* excluding leaf nodes that correspond to top-level signatures). For instance, the width of the hierarchy graph shown in Figure 5.2 is 4

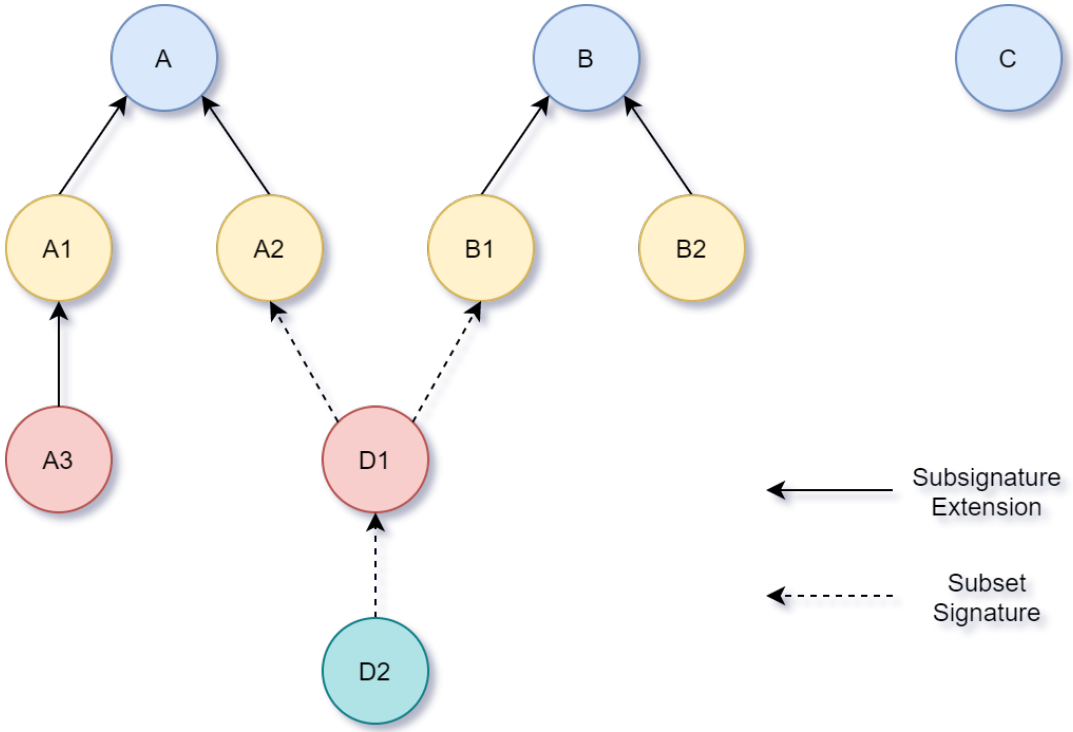

Figure 5.4: Subset Hierarchy Graph

(we do not count the `C` leaf node since it is a top-level signature).

The **subset hierarchy graph** is an augmentation of the extension hierarchy graph obtained by adding subset nodes to the existing graph. We extract subset signature declarations from the model and add nodes to the graph corresponding to the subsets and connect them with an edge. Alloy does not allow the creation of subsignature extensions that extend a subset. Hence, we do not need to account for any additional extension nodes. Once the subset hierarchy graph is built, we compute its depth defined as the longest downward path between a superset and a subset leaf node. The depth of the subset hierarchy graph shown in Figure 5.4 is 2 (the longest downward path is the one between `B1` and `D2`). We do not compute the width of the subset hierarchy graph because subsets can overlap. For instance, in Figure 5.4 `D1` is a subset node of `A1` and `B1` because it is declared as a subset of the union `A1 + B1` and thus there is an overlap of the subsets of `A1` and `B1`.

We report the predominant and typical use criteria for the depth and width of the extension hierarchy graph as well as the depth of the subset hierarchy graph for all models in our corpus. We also present the percentile distribution for the collected depth and width

values.

Measure	**Predominant Use** (Mode)	**Typical Use** (Median)
Depth	1	1
Width	2*	4

Table 5.5: Depth and Width of Extension Hierarchy Graphs

Measure	**12.5th**	**25th**	**50th**	**75th**	**87.5th**	**Common Range**
Depth	0	0	1	1	1	$[0, 1]$
Width	0	1	4	12	30	$[0, 30]$

Table 5.6: Percentile Distribution of Extension Hierarchy Graph Depth and Width

Results: Tables 5.5 and 5.6 show the predominant and typical use values and percentile distribution of the extension graph depth and width respectively. We find that the predominant and typical value for the extension hierarchy graph are both one. Hence, in a typical Alloy model, signatures are not extended beyond a single level. The common range of extension hierarchy graph depth is $[0, 1]$ which means that 75% of depth values are either zero or one. Overall, extension hierarchy graphs in Alloy models are fairly shallow. The non-zero predominant use value for the width of the extension hierarchy graphs is two whereas the typical use value is four. The percentile distribution of the width exhibits a significant amount of variability with the common range being $[0, 30]$. We conclude that 75% of all width values fall between zero and thirty.

Measure	**Predominant Use** (Mode)	**Typical Use** (Median)
Depth	1*	1*

Table 5.7: Depth of Subset Hierarchy Graphs

Measure	**12.5th**	**25th**	**50th**	**75th**	**87.5th**	**Common Range**
Depth (non-zero)	1	1	1	1	1	$[1, 1]$

Table 5.8: Percentile Distribution of Subset Hierarchy Graph Depth

Tables 5.7 and 5.8 show the predominant and typical use values and percentile distribution of the subset hierarchy graph depth respectively. We present the non-zero criteria

and percentile distribution given that the all-inclusive values were zeros due to the scarcity of subset signatures in Alloy models. The non-zero predominant and typical values for the depth of subset hierarchy graphs are both one, which indicates that typical Alloy models rarely use subset signature hierarchies that span over more than one level. The non-zero common range for the depth of subset hierarchy graphs is $[1, 1]$ which indicates that 75% of all non-zero depth values are one.

Findings: We conclude that:

- Developers of Alloy optimizations do not need to ensure that their optimizations scale favorably for deep extension hierarchies since deep hierarchies are a rare occurrence in Alloy models.
- Creating subsets of a subset is a rare practice in Alloy and thus optimizations for the language do not necessarily need to account for this particular case of set hierarchy.
- The shallow set hierarchies in Alloy models are another argument in favor of a type system in the Alloy language.

RQ# 25: How connected are the sets in Alloy models?

Motivation: Signatures in Alloy can be connected through signature declarations, fields, relations and formulas. The object-oriented community coined the term **cohesion** to measure the degree of connectedness between the components of a program [32]. Highly cohesive programs are often desirable as they are generally easier to maintain and promote encapsulation. The LCOM or Lack of Cohesion in Methods metric is used to assess the cohesion of a program by counting the number of strongly connected components in each class [26]. A highly cohesive class has only one such strongly connected component *i.e.,* the value of the LCOM metric is one. Inspired by the LCOM metric, we develop the Signature Connectedness Graph (SCG), a construct that measures the degree of connectedness of signatures in an Alloy model. The SCG metric allows us measure the cohesion of an Alloy model. We define the cohesion of an Alloy model as the degree of connectedness between its components. Exploring the connectedness of signatures in Alloy also provides us with valuable insight that can be used when developing optimizations for the language.

Approach: The **Signature Connectedness Graph** or **SCG** is a measure of the number of strongly connected components of a model's signature graph. This metric is a good indication of how often a signature references another signature. In a signature graph, two signatures are connected if:

- A signature extends another one.
- A signature is a subset of another one.
- A signature's field uses another signature
- The formulas within a signature make reference to another signature or another signature's fields.

We assess the number of strongly connected components in the graph as the value of the SCG metric. An SCG of one indicates that every signature is connected in some way to all other signatures. An SCG of zero occurs when there are no signatures in a model, which can be characterized as a bad modeling practice. A SCG>1 indicates that there exists a number of signature sub-graphs that are not connected.

We gradually build the SCG over multiple phases. We start by extracting subsignature extension declarations from the model. We then iterate over the declarations and identify the names of the extensions as well as the parent signature. We add the extension and parent vertices to the graph and create an edge between each extension and its parent. We also account for subsignature extensions declared using `enum` . We repeat this process with subset signature declarations and augment the SCG with additional vertices representing subset signatures and their parent signatures. We also add edges between the subset and superset vertices. Next, we add any remaining top-level signatures that are not already in the SCG.

In the next phase, we extract field declarations from the body of signatures. If a field references one or multiple other signatures, we add vertices for these referenced signatures to the SCG (if they are not already in it) and create edges between the referenced signatures and the signature under which the field is introduced. In the final step of the SCG building process, we inspect the formulas in the signature fact blocks and augment the SCG accordingly. If a formula contains a reference to another signature, we create an edge between the referenced signature and the signature that contains the formula. Referenced signatures that are not already in the SCG are added before creating the aforementioned edge. Similarly, if a formula in a signature fact block references the field of another signature, we create an edge between the signature that contains the relation and the one that contains the formula. For example, Figure 5.5 shows a number of signature declarations. `A` (on line 1) is a top-level signature with a field `f1` on line 2 that references the signature `B1` declared as a subset of `B` on line 6. The signature `B` is an top-level signature declared on line 5. `A1` and `A2` are subsignature extensions of A declared on line 4. The facts block associated with `C` contains a reference to the field `f1` of `A`. Figure 5.6 contains

the SCG corresponding to the signature declarations in Figure 5.5. The graph contains six vertices: `A`, `A1`, `A2`, `B`, `B1` and `C`. The colored bidirectional arrows indicate the four types of connections that can exist between two signatures. The edge that connects `A` to `A1` and `A2` indicates that `A1` and `A2` are subsignature extensions of `A`. `A` is also connected to `B1` with a field connection arrow since the `f1` field of `A` references `B1`. Additionally, `B1` is connected to `B` with subset signature connection. Lastly, the formula connection arrow between `A` and `C` exists due to the `f1` (which is a field of `A`) reference in the facts block of `C`. When examining the SCG in Figure 5.6, we immediately notice that there exists a path between any two signature vertices in the graph *i.e.,* there is only one strongly connected component that spans the entire graph. Hence, the SCG metric value is one.

```
1   sig A {
2    f1 : B1
3   }
4   sig A1, A2 extends A {}
5   sig B {}
6   one sig B1 in B {}
7   sig C {} {no iden & f1}
```

Figure 5.5: Alloy Signature Declarations

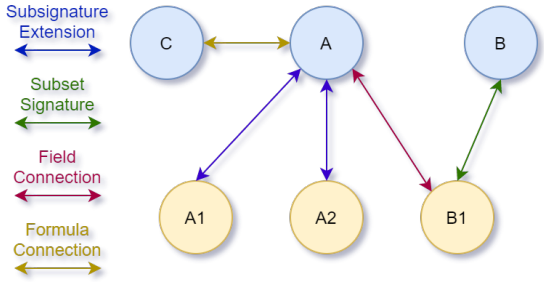

Figure 5.6: SCG = 1

```
1   sig A {
2    f1 : B1
3   }
4   sig A1, A2 extends A {}
5   sig B {}
6   one sig B1 in B {}
7   sig C {}
```

Figure 5.7: Alloy Signature Declarations

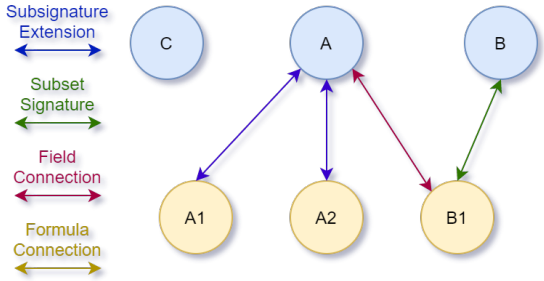

Figure 5.8: SCG = 2

The signature declarations in Figure 5.7 are identical to the ones if Figure 5.5 with the exception of the removal of the formula in the facts body of `C`. Figure 5.8 shows the SCG corresponding to the signature declarations in Figure 5.7. Note that `C` is no longer connected to `A` since the facts body of `C` does not reference `f1` anymore. The graph in Figure 5.8 contains two distinct strongly connected components: the first one consists of

a single vertex (`C`) and the second one consists of the group of vertices `A`, `A1`, `A2`, `B` and `B1`. Consequently, the SCG metric value of the graph in Figure 5.8 is two.

Computation	**Predominant Use** (Mode)	**Typical Use** (Median)
SCG	1	1

Table 5.9: SCG Metric Value

Computation	**12.5th**	**25th**	**50th**	**75th**	**87.5th**
SCG	1	1	1	1	2

Common Range: $[1, 2]$

Table 5.10: Percentile Distribution of SCG Metric Values

Once the SCG is built, we proceed to compute and report the metric value by counting the number of strongly connected components in it. We present the predominant and typical use SCG values as well as a percentile distribution and a common range.

Results: Tables 5.9 and 5.10 show the use criteria values and percentile distribution of the SCG metric respectively. The predominant and typical use values for the SCG metric are both one. The percentile distribution shows that 62.5% of SCG values are exactly one. The common range $[1, 2]$ indicates that 75% of all SCG values are either one or two. We come to the conclusion that the overwhelming majority of Alloy models have an SCG of one and feature a strongly connected set of signatures.

Findings: We believe that:

- Alloy models are highly cohesive given that the overwhelming majority of models in our corpus had an SCG metric value of one.
- Modelers may benefit from having access to the SCG to understand their models. Language and tool designers may want to consider creating a tool that generates the SCG graph and metric for Alloy models.

RQ# 26: What kind of sets is quantification applied to?

Motivation: Alloy allows modelers to create quantified constraints that apply to one or more quantified variables. Quantifiers in Alloy include `all`, `some`, `no`, `lone` and `one`.

We examine the sets associated with quantified variables and classify them based on their hierarchical status (top-level, subset or extension). We also examine the use of quantified set expressions *i.e.,* quantification is applied to an expression that binds two or more sets, which could be signatures or fields. Subsets and subsignature extensions make type checking significantly more challenging and are one of the reasons that Alloy has a single universe and little type checking. However, if most quantified sets are not subsets or extensions, then a type checking system may be a worthwhile investment for Alloy and subsets can be expressed via set membership.

Approach: We extract from the model the names of all top-level signatures, subset signatures and subsignature extensions. We then extract sets names from quantified expressions and classify them accordingly. We are only able to classify sets that are declared in the model and not those imported from external modules. We also extract and tally up the total number of quantified set expressions. We report the percentage distribution over the different kinds of quantified variables.

Kind	**Distribution**
Top-Level	56.6%
Subset	0.7%
Extension	19.9%
Set Expression	22.8%

Table 5.11: Quantification by Variable Kind

Results: Table 5.11 shows the results of this research question. Top-level sets constitute the vast majority (56.6%) of sets used in quantified expressions. Subsignature extensions are the second most prominent type of sets used in quantification (19.9%). Subset signatures account for 0.6% of the total number of sets used in quantification which leads us to believe that quantification over subsets is rare in Alloy models. Lastly, set expressions account for 22.8% of all sets used in quantified expressions.

Findings: Our results suggest that:

- Given that most quantified sets are top-level signatures, a type system may be a worthwhile addition to Alloy. Language and tool designers who may want to consider adding a type checking mechanism to Alloy.

RQ# 27: How often are signatures used as structures?

Motivation: Alloy does not provide a syntax for structures that are just containers (records). Instead, a separate set can be introduced to act as a mapping to the elements contained in the structure. An alternative way to mimic structures in Alloy is to use relations of high arity (*i.e.,* large tuples), which would reduce the number of sets in the model. However, relations of high arity generally cause poor performance in analysis plus it is not possible to use the transitive closure operator on relations of arity greater than two. If signatures are often used as structures, then Alloy language designers might consider providing support (in syntax and analysis) for structures as is found in other declarative modeling languages such as TLA+, B and DASH.

Approach: A set is being used as a structure if it is only used as an index to the elements of the structure and never used by itself. We identify sets that are only used to access fields via a join operator in formulas. We start by extracting all signature names and identifying the box or dot join expressions involving each distinct signature and one of its fields. Expressions that contain an application of the transitive closure operators (*e.g.,* `A.^f1` or `A.*f1`) are excluded from this count. We report the total number of signatures used as structures for each model.

Results: We find that 0.4% of all signatures are used only as structures. The typical use criterion is 0, which leads us to believe that a typical Alloy model does not contain a signature used as a structure. We conclude that signatures are rarely used as structures.

Findings: We find that:

- Even though other declarative modeling languages such as TLA+ have structures, this construct does not appear to be needed in Alloy.

RQ# 28: Are Alloy modelers creating abstract signatures with no children?

Motivation: Abstract signatures are generally used for the sole purpose of creating subsignature extensions from them. However, Alloy does not prevent users from creating abstract signatures with no extensions. In this case, the abstract signature behaves like a regular signature. This practice is discouraged and can be categorized as a bad modeling practice.

Approach: We extract from the model all the names of parent signature as well as abstract signatures. We then cross-reference the two lists to identify abstract signatures that are not parent signatures. The identified signatures are declared abstract but they are not extended in the model. We report the percentage of abstract signatures without any children across

all models and across models that are not declared as modules. We consider the subset of models that are not declared as modules to account for the fact that abstract signatures declared in a user-created module may be extended in another file that import the module.

Results: The percentage of abstract signatures with no children comes up to 2.6% out of the total number of signatures (or 31.1% of abstract signatures) across all models. When considering only models that are not declared as modules, we find that abstract signatures with no children constitute 0.5% of all signatures (6.3% of abstract signatures) in this subset of models. These results confirm that creating abstract signatures with no children is a rare practice among Alloy modelers given that most abstract signatures with no children are declared in modules and thus are probably being extended in other models.

Findings: We conclude that:

- Abstract signatures with no children do not appear to be a problem that needs to be addressed by language and tool designers.

RQ# 29: Are modelers creating abstract signatures with no fields?

```
1  abstract sig A {}
2  sig A1, A2 extends A {}
3  pred at_least_one_A  {some A}
4  run at_least_one_A for 5 A
5
6  sig B1, B2 {}
7  pred at_least_one_B {some B1 + B2}
8  run at_least_one_B for 3 B1, 2 B2
```

Figure 5.9: Abstract Signatures without Fields

Motivation: Abstract signatures are used to their full extent when they have fields that can be inherited by their subsignature extensions. However, abstracts signatures without fields can still be created in Alloy. While their utility is reduced compared to abstract signatures with fields, they still offer some advantages. Daniel Jackson notes that abstract signatures without fields can make a model shorter and easier to modify in the future [10]. For example, Figure 5.9 shows how abstract signatures without fields can still be beneficial to modelers. Signature `A` on line 1 is declared as an abstract signature (without fields) with two subsignature extensions `A1` and `A2` on line 2. Since `A` is abstract, we know that `A = A1 + A2`. Thus, any time we need to reference the union of `A1` and `A2`, we can simply use

`A` as shown in the predicate `at_least_one_A` on line 3. Should the model be modified in the future to include a third subsignature extension `A3` of `A`, the union reference would not need to be updated. If we did not make use of an abstract signature with no fields, we would have to explicitly reference the union as shown in predicate `at_least_one_B` on line 7. These union references would have to be manually updated in the future if the modeler was to add more subsignature extensions of `A`. Using abstract signatures with no fields also allows modelers to take advantage of the scope inference mechanism of Alloy. The command on line 4 only sets the scope for `A` and lets the Alloy Analyzer infer the scopes of `A1` and `A2` whereas the command on line 8 explicitly mentions the scopes of `B1` and `B2`. While the advantages offered by abstract signatures with no fields are significant, they do not tap into the inheritance capabilities of Alloy. We explore the use of abstract signatures without fields to help educators better understand how modelers are using abstract signatures in their models

Approach: We extract from the model all abstract signature declarations and classify them as either abstract signatures with fields or abstract signatures without fields. We present the percentage distribution of abstract signatures between these two categories.

Construct	Distribution
Abstract Signatures with Fields	31%
Abstract Signatures without Fields	69%

Table 5.12: Distribution of Abstract Signatures with and without Fields

Results: Table 5.12 shows the results of this research question. We find that abstract signatures without fields account for 69% of all abstract signatures whereas abstract signatures with fields account for 31% of all abstract signatures. It is evident that modelers are using abstract signatures without fields extensively.

Findings: We find that:

- Educators should ensure that abstract signatures with no fields are being used for purposeful reasons (as described above) rather than just to deepen the set hierarchy.

5.4 Formulas

Formulas in Alloy are used to describe constraints about the model. Formulas differ significantly depending on the modeling style used. In this section, we examine the different

styles of writing formulas.

RQ# 30: How often do modelers use the different styles of writing formulas?

Motivation: We consider the three styles for writing formulas described at the beginning of Jackson's book on Alloy [37]. Figure 5.10 shows the same formula expressed in three different styles. The formula states that the relation `f` maps all elements of `A` to some element in `B1`. The three formula styles are defined as follows:

- In the **predicate calculus style** (shown on line 1 of Figure 5.10), quantifiers over variables are used along with Boolean operators (but no set operators).
- In the **relational calculus style** (shown on line 2), expressions denote relations, and multiplicity operators on relations are used to accomplish quantification.
- In the **navigation expression style** (shown on line 3), expressions denote sets and quantification can be used.

Predicate calculus is commonly used in comprehension expressions to create a set or relation from a constraint. It is also used for subtle constraints since it often matches the formulation of the constraint in natural language. The relational calculus style results in very concise formulas. The three styles do not have equivalent expressive power. The navigational style is the most expressive among them because the predicate and relational calculus styles lack transitive closure and quantifiers respectively. Some formulas written in Alloy may not fit into any of the three modeling styles. By answering this question, we help educators determine how much to emphasize the distinction between the different styles in teaching.

```
1  all a1,a2:A| a1->a2 in f2 implies a1 != a2 // predicate calculus
2  no iden & f2                               // relational calculus
3  no a: A | a = a.f2                         // navigation expression
```

Figure 5.10: Three Formula Modeling Styles

Approach: We extract formulas contained within the body of facts, predicates, functions and assertions. The collected expressions are then filtered to exclude sub-expressions. The processed collection of formulas includes only top-level expressions. Next, we perform post-processing on each expression to identify the style used. Formulas containing a quantified variable and field names without any occurrences of set operators fall under the predicate

calculus style category. If a formula does not contain a quantified variable but references a relation name with or without set operators, then it is classified as a relational calculus formula. Finally, if an expression contains a quantified variable and set operators, then it falls under the navigation expression style. Predicate and function calls in a formula are ignored for this classification. We count the number of formulas belonging to each formula style. After processing all the formulas in a file, we classify the model as one of the following six categories: pure predicate calculus, pure relational calculus, pure navigation expression, dominant predicate calculus, dominant relational calculus, or dominant navigation expression. If all the formulas in a model fall under one modeling style, then the model will have a pure label corresponding to that formula style. Otherwise, the model gets labeled with the dominant writing style *i.e.,* the style matching the largest number of formulas in that model.

Model Classification	**Distribution**
Pure Relational Calculus	33.9%
Dominant Relational Calculus	43.5%
Pure Navigation Expression	4.9%
Dominant Navigation Expression	16.9%
Pure Predicate Calculus	0.3%
Dominant Predicate Calculus	0.5%

Table 5.13: Model Classification by Formula Style

Results: Table 5.13 shows the relational calculus formula style is the most-used style across the pure and dominant categories whereas predicate calculus is the least common style. We conclude that Alloy modelers prefer relational calculus and tend to avoid predicate calculus. Navigation expression is a popular formula style with modelers likely when the constraint is too complex to be expressed with relational calculus. The ubiquity of the relational calculus style highlights the usual goal of overall simplicity in Alloy modeling because the vast majority of formulas are expressed with the most concise style.

Findings: We find that:

- Educators may want to consider the frequency of use of the different formula modeling styles when deciding which modeling style to teach first or emphasize.
- A user study to determine how novice modelers use the different formula styles compared to experienced modelers may be of interest.

5.5 Scopes

Alloy models are bounded *i.e.,* they must have a specified maximum possible size. A **scope** is a bound on the size of signature sets and can be changed for each command query. If the maximum scope for a set is not specified, the Alloy Analyzer will assume that each top-level signature is limited to three elements. In this section, we explore how modelers are setting scopes in their models, the kinds of signatures that have set scopes as well as scope-setting practices that are discouraged.

RQ# 31: How are scopes chosen in Alloy models?

Motivation: For analysis, the Alloy Analyzer chooses a default scope for sets that have not been assigned a scope in a query. If a modeler does not include a scope, it can indicate either that the default scope is adequate for their analysis or that they do not know how to choose a reasonable scope. We also examine how often exact scopes are used (rather than all scopes less than or equal to the set scope). An exact scope limits the number of instances that need to be checked. Hence, an exact scope is likely to reduce the analysis time, but produces a less general result if the model is unsatisfiable. But there are some sets that are inherently of an exact scope (*e.g.,* colors of a traffic light).

```
1   sig A {}
2   sig A1, A2 extends A {}
3   pred show {}
4   run show for 2 A
```

Figure 5.11: Scope Derivation Case 1

```
1   sig A {}
2   sig A1, A2, A3 extends A {}
3   pred show {}
4   run show for 6 A, 1 A1, 3 A2
```

Figure 5.12: Scope Derivation Case 2

```
1   sig A {}
2   sig A1, A2, A3 extends A {}
3   pred show {}
4   run show for 1 A1, 3 A2, 2 A3
```

Figure 5.13: Scope Derivation Case 3

Approach: The answer to this research question is complicated by the fact that the scope of a signature may be derived from the scope of another signature. Scope derivation stems from the set hierarchy. For instance, if a top-level signature with a set scope has multiple child subsignature extensions with no scopes associated with them, the scopes of the extensions are derived to be equal to that of the parent signature. Figures 5.11 and 5.14 illustrate this case of scope derivation. The top-level signature `A` is extended

with two subsignature extensions `A1` and `A2`. The `run` command on line 4 sets the scope of `A` to two. The scopes of `A1` and `A2` are inferred to be two as well. Similarly, if a top-level signature has a set scope and all its subsignature extensions (mutually disjoint) except for one have scopes associated with them, then the missing scope of the extension can be derived. This second case of scope derivation is shown in Figures 5.12 and 5.15. The command query on line 4 sets to the scope of `A`, `A1` and `A2` to six, one and three respectively. The scope of `A3` is inferred to be two.

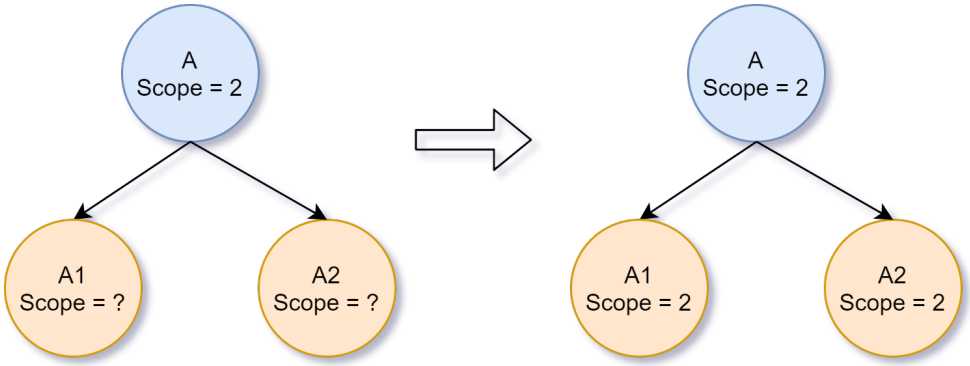

Figure 5.14: Scope Derivation Case 1

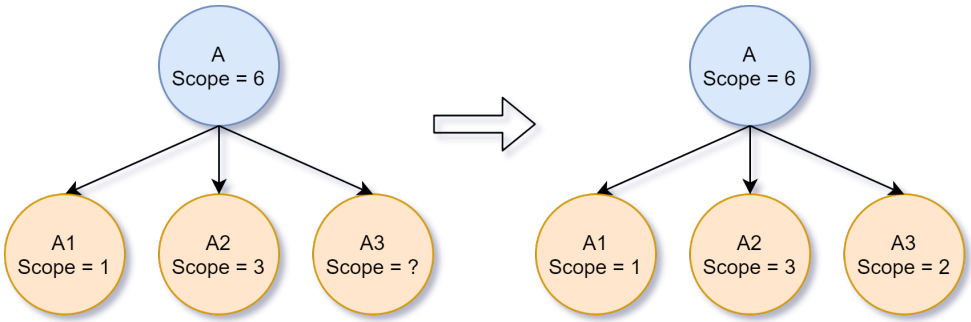

Figure 5.15: Scope Derivation Case 2

On the other hand, if the top-level signature does not have a set scope, but its children subsignature extensions have scopes associated with them, then the parent signature has a derived scope equal to the sum of the scopes of its child extensions as shown in Figures 5.13 and 5.16. The command on line 4 sets the scope of `A1`, `A2` and `A3` to one, three and two respectively. The Alloy Analyzer computes the scope of `A` by summing the scopes of `A1`, `A2` and `A3` and `A` gets a scope of six.

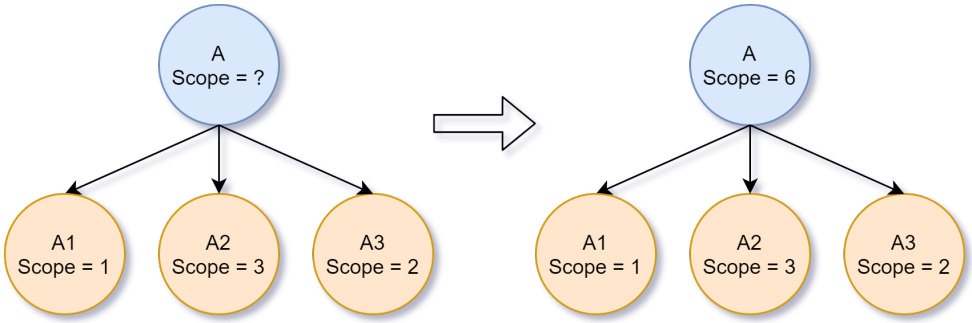

Figure 5.16: Scope Derivation Case 3

We devise six distinct categories for the scope of a signature:

- **Set exact**: The scope is explicitly set in the run/check command using the keyword `exactly`.
- **Set non-exact**: The scope is set in the run/check command without using the `exactly` keyword.
- **Derived exact**: The scope is not explicitly set, but derived to be exact.
- **Derived non-exact**: The scope is not explicitly set, but derived to be non-exact.
- **Model Exact**: The scope is not explicitly set, but required to be an exact value by the model (*e.g.*, the use of `one` in a signature declaration).
- **Default non-exact**: The scope is not explicitly set and cannot be derived so it is assigned a default non-exact scope.

The Alloy Analyzer sets the scopes for all sets using explicitly set scopes, derived scopes, and default values for scopes. We extract and classify the scopes for all signatures in individual run and check commands in all the models.

Results: Table 5.14 shows that half of all run and check queries fall in the default non-exact category. Model exact is the second most populous category coming in at 18.2%. The scarcity of exact scopes compared to non-exact scopes is evident when examining the results. We hypothesize that modelers may not be familiar with the `exactly` keyword and consequently are not using it abundantly.

Findings: The findings of this research question are as follows:

Scope Category	Distribution
Default Non-Exact	51.5%
Model Exact	18.2%
Set Non-Exact	15.6%
Set Exact	8.8%
Derived Non-Exact	4.8%
Derived Exact	1.2%

Table 5.14: Scopes Categories Across All Queries and All Models

- Most Alloy modelers formulate queries without specifying scopes in them. We also find that modelers prefer setting an upper bound for sets as opposed to dictating an exact size.
- Educators are encouraged to explain when exact scopes are appropriate.
- Further studies should be conducted to explore the relationships between scopes, Jackson's generator axioms [37] and the significance axioms discussed in [30]. Generator axioms are used to ensure that instances where increasing the scope would satisfy an assertion are not erroneously produced as counterexamples. Significance axioms ensure that the scope is large enough to be interesting. The scope only increases linearly to the finite number of operations.

RQ# 32: Are scopes being set at the top level or subset/subsignature extension level?

Motivation: Scopes can be set for top-level signatures, subset signatures and subsignature extensions. Modelers can set these scopes in a command query or in a signature declaration. In this research question, we examine the type of signatures that have scopes assigned to them as well the different levels (query or signature) at which these scopes are being set. By inspecting scope levels, we can help educators get a better understanding of how modelers are setting scopes in their Alloy models.

Approach: We extract signature names from `run` and `check` commands and then classify them as top-level signatures, subset signatures or subsignature extensions. We then tally up the number of each signature type. Similarly, we extract the names of signatures declared with a multiplicity (*e.g.,* `lone` and `one`) and categorize them. We exclude signatures declared with multiplicity `some` since they do not constrain the set size to a specific value and thus do not fall under the same category as commands and the multiplicities `lone`

and `one`. We report data summary criteria for top-level signatures, subset signatures and subsignature extensions at the command and signature levels.

Kind	Distribution
Top-level	83.7%
Subset	0.0%
Extension	16.3%

(a) Commands

Kind	Distribution
Top-level	4.4%
Subset	0.2%
Extension	95.4%

(b) Signatures with Multiplicity `one`

Kind	Distribution
Top-level	0.0%
Subset	0.8%
Extension	99.2%

(c) Signatures with Multiplicity `lone`

Table 5.15: Scope Levels

Results: The results of this research question are summarized in Table 5.15. At the command level, the vast majority of scopes (83.7%) are attributed to top-level signatures. The remaining 16.3% of scopes at the command level are subsignature extensions. We did not find any subset signatures with a scope set at the command level. Signature kind distribution is fairly similar for signatures with multiplicity `one` and `lone`, with subsignature extensions being the dominant type of scope assignments. We did not identify any top-level signatures with multiplicity `one`.

Findings: We come to the following conclusions:

- Subsignature extensions account for the vast majority of signatures with multiplicity `one` and can be replaced with a dedicated scalar construct. Language and tool designers may want to add a scalar construct to Alloy.
- Subsignature extensions constitute the vast majority of signatures with multiplicity `one`. Thus, educators are encouraged to highlight `enum`s as a means to create ordered subsignature extensions with multiplicity `one`.

RQ# 33: How often is the ordering module applied to a set with a non-exact scope?

Motivation: The ordering module can be used to place an ordering on a parameterized signature. The ordering module forces the signature on which it was instantiated to be exact. The Alloy Analyzer does not explicitly warn modelers against using a non-exact scope with an ordered signature. The implicit exact constraint can lead to unexpected and erroneous results.

```
1  open util/ordering[C]
2
3  abstract sig A {}
4  sig B extends A {}
5  sig C extends A {}
6
7  pred show {}
8  run show for 7
```

Figure 5.17: Model from Stack Overflow Question [11]

Figure 5.17 shows an Alloy model posted on Stack Overflow [11] that showcases the issues that can be caused by using a non-exact scope with an ordered signature set. In this model, `A` is an abstract signature with two subsignature extensions `B` and `C`. The predicate `show` does not contain any constraints. The `run` command on line 8 uses a set non-exact scope of 7. If the signature `C` was not an ordered set, then `A`, `B` and `C` would be given a non-exact scope *i.e.,* an upper bound of 7. In any instance of the model, the sum of the elements in `B` and `C` is at most 7. However, since `C` is an ordered set, it is forced to have an exact scope of 7. This exhausts the scope quota for signature `A`, which leaves signature `B` with an implicit scope of 0. Consequently, no instances of the model will have elements in `B`, which may not be the intention of the modeler. We examine the use of the ordering module on sets with a non-exact scope to determine if this bad modeling practice is prevalent among modelers.

Approach: We extract from the model the set names that have an ordering applied to them using an `open` statement. Next, we extract from command queries all sets that have a non-exact scope. We include all three non-exact scope categories (default non-exact, set non-exact and derived non-exact). Finally, we cross-reference the lists of sets to identify ordered sets with a non-exact scope. If a signature set is used in multiple command queries and thus falls under multiple scope categories it is counted once for the total number of ordered sets with a non-exact scope. However, we count duplicates multiple times when computing the percentage distribution across commands in all models.

Scope Category	Distribution
Set Non-Exact	84.4%
Default Non-Exact	14.9%
Derived Non-Exact	0.7%

Table 5.16: Scope Category Distribution Among Ordered Sets

Results: We find that modelers are frequently applying the ordering module to a set with a non-exact scope given that 54.8% of all ordered sets have a non-exact scope. Table 5.16 shows the distribution of ordered sets across the non-exact scope categories. The vast majority of ordered sets with a non-exact scope fall under the set non-exact scope category (84.4%) followed by the default non-exact scope category (14.9%). Only 0.7% of ordered sets with a non-exact scope fall under the derived non-exact category. Given that set non-exact is the dominant scope category, we can say with a certain degree of confidence that modelers are explicitly setting non-exact scopes for ordered sets. We hypothesize that these results are an underestimate of the actual frequency of occurrence of this practice.

Findings: We conclude that:

- Educators are encouraged to highlight this bad modeling practice and explain its repercussions on the command results.
- Language designers may want to update the Alloy Analyzer to allow it to generate a warning when the ordering module is applied to a set with a non-exact scope.

RQ# 34: How are scopes set for integers in Alloy?

Motivation: In Alloy, integer scope specifications determine the maximum bit-width for integers. For instance, a command containing the scope `6 Int` assigns to the signature Int the range of integers from -32 to + 31. The default integer scope in Alloy is 4 so only integers in the range $[-8, 7]$ are considered during the instance generation. Setting an integer scope that is too low may result in an overflow that causes valid counterexamples to be missed by the analyzer. Setting a scope that is too high may negatively affect solving time. We seek to determine what scope values modelers assign to integers.

Approach: We extract and tally up all command queries containing an integer scope specification. We compute the predominant and typical use criteria values as well as the

Integer Scope	**Predominant Use** (Mode)	**Typical Use** (Median)
Value	5	5

Table 5.17: Integer Scope Values

percentage distribution of integer scope value and present the percentage of integer scope out of the total number of scoped sets in commands.

Integer Scope	**12.5$^{\text{th}}$**	**25$^{\text{th}}$**	**50$^{\text{th}}$**	**75$^{\text{th}}$**	**87.5$^{\text{th}}$**
Value	3	4	5	6	8

Common Range: $[3, 8]$

Table 5.18: Percentile Distribution of Integer Scope Values in Commands

Results: We find that integer scope specifications account for 8% of the total number of scoped sets in commands which indicates that setting integer scopes in Alloy models is not uncommon. Table 5.17 shows the predominant and typical integer scope criteria. A typical Alloy model sets the scope of integers to 5 as shown by the typical use criterion. The most common integer scope value is also 5 as shown by the predominant use criterion. Table 5.18 shows the percentile distribution of integer scope values. We find that the common range for integer scopes is $[3, 8]$ *i.e.,* 75% of integer scope values fall in that range.

Findings: Our findings are listed below:

- Given that the default scope for integers in Alloy is 4, some modelers may be making their problems too large by specifying integer scopes that are higher than needed.
- Language and tool designers may want to reconsider the default scope for integers in Alloy given that modelers often specify higher integer scopes than the default scope.
- Educators are encouraged to familiarize students with proper integer scope use and help them find alternative ways of modeling problems with numeric constants without using integers. (See Section 5.2)

5.6 Summary

In this chapter, we presented a series of research questions that examine how the constructs of the Alloy language are used. These "deeper-level" questions are split into five subsections: Modules, Integers, Sets, Formulas and Scopes. In Section 5.1, we explore the use of user-created and library modules. Section 5.2 investigates the use of integers and the possibility of substituting integers for alternative constructs. Next, we examine the use of sets in Section 5.3 and perform an extensive study of the relationships and hierarchies that bind them. In Section 5.4, we classify formulas in Alloy models based on the writing used to formulate them. Lastly, Section 5.5 explores the scope-setting practices of Alloy modelers. We present a summary of our findings split into three categories: findings for language and tool designers, findings for educators and findings for optimization developers.

Findings for Language and Tool Designers:

- Alloy language and tool designers may want to develop model management IDEs to help modelers manage imported user-created modules.
- Some library modules are rarely used and hence language designers may want to drop them in future versions of Alloy.
- Language designers may want to add a built-in order construct to represent numeric constants.
- 55.3% of all integers used in fields do not need to be integers and could be a set with a linear ordering. Language designers could modify the Alloy Analyzer to allow it to warn users about integer constants that are only used as a linear order.
- The Alloy Analyzer could notify modelers about integer uses that can be replaced with an application of the ordering module.
- Modelers are using the set cardinality operator to set the size of signature sets. Given that this practice is inefficient in analysis, tool designers are encouraged to add a warning that discourages its use and provides alternative methods to specify the signature set size (multiplicity keywords, command queries).
- Language and tool designers may want to give modelers access to the SCG to allow them to gauge the connectedness of their models.
- Language and tool designers may want to add a scalar construct to Alloy.

- Language and tool designers are encouraged to add a type system to Alloy along with type checking mechanisms.
- A dedicated structure/record construct is not needed in Alloy.
- Abstract signatures are almost always extended with subsignature extensions.
- The ordering module is commonly applied to sets with a non-exact scopes. We recommend that tool designers discourage this practice by warning modelers when they use a non-exact scope with an ordered set.
- Language and tool designers may want to reconsider the default scope for integers in Alloy given that modelers often specify higher integer scopes than the default scope.

Findings for Educators:

- Modelers are not using the ordering module to its full extent given the profuseness of integers used solely to impose a linear ordering. Educators are encouraged to promote the use of the ordering module.
- We encourage educators to highlight the importance of using multiplicity keywords and command queries to set the size of signature sets in lieu of the set cardinality operator because setting the size of a signature using the set cardinality operator leads to longer solving times.
- Educators may want to consider the frequency of use of the different formula modeling styles when deciding which modeling style to teach first or emphasize.
- `enum`s are an underutilized construct in Alloy. Educators are encouraged to highlight `enum`s as a means to create ordered subsignature extensions with multiplicity `one`.
- Modelers tend to use non-exact scopes far more frequently than exact scopes. Educators are encouraged to explore the different scope categories with student modelers and highlight the performance gains that can be achieved by using exact scopes.
- Educators should discourage the use of non-exact scopes with an ordered set.
- Integer scopes should be explained in greater detail. Educators should also encourage students to find alternative ways of modeling problems with numeric constants that do not involve integers.

Findings for Optimization Developers:

- Developers are encouraged to create optimizations on analysis for the ordering module and for integers in Alloy because they are used frequently.
- Optimization developers are encouraged to internally transform integer applications used to impose a linear order to ordered sets that take advantage of the ordering module.
- We recommend that developers invest in the development of Alloy optimizations that convert applications of the set cardinality operators that specify the size of a signature set to command queries or signature declarations with a multiplicity keyword because specifying the size of a signature using the set cardinality increase model complexity and solving time.
- Given the fairly shallow depth of extension set hierarchy graphs in Alloy models, optimizations do not necessarily need to scale favorably for deep extension hierarchies.
- Creating subsets of a subset is a fairly rare practice in Alloy. Hence, optimization developers do not necessarily need to account for this particular instance of set hierarchy when developing their tools and engines.
- The shallow set hierarchies in Alloy models are another argument in favor of a type system in the Alloy language.

Chapter 6

Analysis Complexity

In this chapter, we explore research questions related to Alloy model features that could affect analysis complexity by increasing solving time. We also examine the use of certain model features such as partial and total functions that can be exploited for optimizations in different solvers. The research questions in this chapter can be used as a basis for future work optimizations in solvers. We discuss the use of second-order operators (set cardinality and transitive closure), partial and total functions and the depth of joins and quantification.

6.1 Second-order Operators

This section explores the use of second-order operators in Alloy models. Searching for instances in models that use second-order features involves extensive expansion of formulas and thus can affect analysis complexity and solving time. If the operators are used profusely in Alloy, developers may want to investigate optimizations with these operators in mind.

RQ# 35: How common is the use of the set cardinality operator in Alloy models?

Motivation: The set cardinality operator (`#`) allows modelers to specify the size of a set consisting of a signature, field or formula. We have previously found in Chapter 5 that set cardinality is being used to specify the size of sets in Alloy models. We assess the frequency of set cardinality operator use in Alloy models to help developers determine whether or not to create optimizations centered around this second-order operator.

Approach: We extract from the model all instances of the set cardinality operator. We

account for predicate and function calls that contain uses of the set cardinality operator *i.e.,* the number of set cardinality operators in a predicate or function is scaled according to the number of calls corresponding to that predicate or function. We report the percentage of models that have at least one set cardinality operator in our corpus of models along with the non-zero predominant and typical use criteria. We also present the non-zero percentile distribution and common range for the number of set cardinality operators in a model.

Construct	**Predominant Use** (Mode)	**Typical Use** (Median)
Set Cardinality (non-zero)	2	6

Table 6.1: Set Cardinality Operator Count in Alloy Models

Construct	**12.5$^{\text{th}}$**	**25$^{\text{th}}$**	**50$^{\text{th}}$**	**75$^{\text{th}}$**	**87.5$^{\text{th}}$**
Set Cardinality (non-zero)	2	2	6	14	26

Common Range: $[2, 26]$

Table 6.2: Percentile Distribution of Set Cardinality Operator Count

Results: We find that 34.8% of all models in our corpus have at least one use of the set cardinality operator. Table 6.1 shows the non-zero predominant and typical use values for the set cardinality operator count. We find that the non-zero predominant value is two whereas the non-zero typical value is six. Thus, a typical Alloy that uses set cardinality contains six uses of this operator. Table 6.2 shows the percentile distribution of the set cardinality operator count. We find that the non-zero common range is $[2, 26]$ which means that 75% of set cardinality operator count are between two and twenty-six. Therefore, we conclude that the set cardinality operator count can vary significantly among models that make use of it.

Findings: Our results suggest that:

- The set cardinality operator is used in a significant percentage of Alloy models.
- Models that use set cardinality usually contain a relatively high number of uses of this operator.
- Developers are encouraged to create solvers and optimizations that address the abundant use of the set cardinality operator in Alloy.

RQ# 36: How common is the use of transitive closure in Alloy models?

Motivation: The transitive closure operator is a unary operator that can be applied a binary relation. When the **non-reflexive transitive closure** operator (`^`) is applied to a binary relation `r`, it returns the result of the following expression:

```
^r = r + r.r + r.r.r + ...
```

When the **reflexive transitive closure** operator (`*`) is applied to a binary relation `r`, it returns the following infinite set:

```
*r = iden + r + r.r + r.r.r + ...
```

which can simplified as:

```
*r = iden + ^r
```

Uses of the transitive closure operators involve extensive expansions of the formulas in the model for solving, which leads us to believe that these operators may impact analysis complexity and solving time. We explore the use of transitive closure operators in Alloy models to help optimization developers decide how much to focus on the development of optimizations that target these operators.

Approach: We extract and tally up the number of reflexive and non-reflexive transitive closure operators in the model. We scale the occurrences of these operators in predicates and functions according to the number of calls made to the these constraint containers. We compute and report the percentage of models that make use of transitive closure operators. We report the non-zero predominant and typical use criteria as well the non-zero percentile distribution and common range. We opt to use the non-zero values given that the all-inclusive values were mostly zeroes and thus were not very informative.

Operator	**PU** (Mode)	**TU** (Median)
Transitive Closure (non-zero)	1	5
Non-Reflexive Transitive Closure (`^`) (non-zero)	1	3
Reflexive Transitive Closure (`*`) (non-zero)	1	6

Table 6.3: Transitive Closure Operators in Alloy Models

Results: We find that 26.8% of models in our corpus contain at least one transitive closure operator. We also find that 19.4% of models contain at least one use of the non-reflexive transitive closure operator whereas 16.2% of models have at least one use of the reflexive transitive closure operator. Table 6.3 shows the non-zero predominant and typical

Operator	Percentage Distribution
Non-Reflexive Transitive Closure	35.1%
Reflexive Transitive Closure	64.9% } = 100%

Table 6.4: Percentage Distribution of Transitive Closure Operators

use values for the transitive closure operators. Models that make use of transitive closure operators typically contain five uses of these operators. Table 6.4 shows the percentage distribution of transitive closure operators between the non-reflexive and reflexive categories. While there are slightly more models that contain use of the non-reflexive transitive closure operator, the applications of the reflexive transitive closure operator (64.9%) significantly outnumber those of the non-reflexive operator (35.1%).

Operator	12.5^{th}	25^{th}	50^{th}	75^{th}	87.5^{th}	CR
Transitive Closure	1	2	5	11	16	$[1, 16]$
Non-Reflexive Transitive Closure	1	1	3	5	8	$[1, 8]$
Reflexive Transitive Closure	1	2	6	11	15	$[1, 15]$

Table 6.5: Non-zero Percentile Distribution of Transitive Closure Operators

Table 6.5 shows the percentile distribution and common range of transitive closure operators. We find in the models that make use of transitive closure operators, 75% of the transitive closure operator counts fall between one and sixteen.

Findings: The results of this research question suggest that:

- Given that 26.8% of models in our corpus contain transitive closure operators and given the relatively substantial number of uses of transitive closure operators in these models, we encourage developers to explore developing optimizations centered around these operators.

6.2 Partial and Total Functions

This section explores the use of total and partial functions in Alloy models. These functions are handled differently by different solvers. For example, KodKod represents total functions as relations with additional constraints, whereas Portus [41] represents total functions as total functions for SMT solving. We include this analysis of the types of field relations used

(partial, total) because tools connecting Alloy to SMT solvers (*e.g.,* El Ghazi *et al.* [28]) can express specialized relations as functions in first-order logic.

RQ# 37: How common are partial and total functions in Alloy models?

Motivation: User-introduced partial functions in Alloy are declared as fields under signatures and can take one of the these two forms:

- `f: e1 -> lone e2`
- `f: lone e`

The set multiplicity keyword `lone` indicates that the each element in the domain is mapped to *zero or one* element in the range, which makes the field a partial function. Note `e1` can be a larger set expression that includes multiple sets. A **total function** is a function that maps every element in the domain to some element in the range. User-introduced total functions in Alloy can take one of the following forms:

- `f: e`
- `f: one e`
- `f: e1 -> one e2`

The set multiplicity keyword `one` is used to indicate that every element in the domain is mapped to *exactly one* element in the range. The first two forms are equivalent given that the default multiplicity for a field in Alloy is `one`. In the third form, `e1` may be a larger set expression that references other sets. The built-in Alloy library modules contain a number of total and partial functions. We explore the use of user-introduced, library and overall (*i.e.,* including user-introduced and library) total and partial function in Alloy models to assess whether or not optimizations and solvers that target these constructs should be developed.

Approach: We count the number of user-introduced and library partial and total functions across all models. We also consider the overall number of partial and total functions which includes user-introduced total and partial functions and those imported from a library module and used at least once. We present the percentage distribution of fields across these different categories.

Results: Table 6.6 shows the percentage distribution of fields among the six categories of partial and total functions. We find that user-introduced partial functions account for

Function	Percentage Distribution
User-introduced Partial Functions	10.7%
User-introduced Total Functions	50.0%
Library Partial Functions	2.1%
Library Total Functions	6.9%
Overall Partial Functions	12.9%
Overall Total Functions	57.0%

of all fields

Table 6.6: Percentage Distribution of Total and Partial Functions

10.7% of all fields whereas user-introduced total functions constitute 50% of all fields. We also find that 58.6% of all user-introduced total functions (*i.e.,* 29.3% of all fields) are over top-level signatures. SMT solvers treat user-introduced total functions over subsets as partial functions. Hence, optimizations that target total functions would be limited to user-introduced total functions over top-level signatures. Library partial functions are significantly less common and account for 2.1% of fields only. Library total functions are used more frequently account for 6.9% of all fields. Overall, partial functions account for 12.9% of all fields whereas total functions account for 57% of all fields. Total functions are clearly prevalent in Alloy models. Partial functions are not as abundant as total functions but still account for a considerable portion of all fields.

Findings: We conclude that:

- Total functions are used extensively in Alloy models. Thus, it is worthwhile for developers to work on improving the analysis methods with a focus on total functions.
- Partial functions are used modestly in Alloy models. Developers could potentially create optimizations that handle these functions.

6.3 Depth of Joins and Quantification

The depth of joins and quantification are factors that are generally considered to make automated verification of finite problems take longer. In this section, we discuss the use the dot and join operators in Alloy and the depth of joins in formulas. We also explore the depth of quantification in Alloy models to assess whether or not developers should focus on improving analysis methods with a focus on nested quantification.

RQ# 38: What the typical depth of joins in Alloy models?

Motivation: The **dot join** `f1.f2` of two relations `f1` and `f2` is the relation obtained by taking every combination of a tuple in `f1` and a tuple in `f2` and including their join, if it exists. The relations `f1` and `f2` do not need to have the same arity. The **box join** operator `[]` is semantically identical to the dot join but its arguments are in a different order and it has different precedence. For instance, the expression `f1[f2]` is equivalent to `f2.f1`. Dot joins bind tighter than box joins *e.g.,* `f1.f2[f3]` is equivalent to `f3.(f1.f2)`. The main reason for the inclusion the box join operator in Alloy is its ability to provide a syntactic means of expressing an indexed lookup in a way that resembles object-oriented programming languages. Deep joins involve extensive expansions of the formulas for solving [1] and could affect solving time and analysis complexity. We explore the use and depth of box and join operators in Alloy models.

Approach: We count individual uses of the dot and box join operators to get a better idea of the frequency of these operators. We also compute the depth of joins by extracting the top-level expressions containing applications of the dot and box join operators and counting the number of expressions bound together by these operators. We report a percentage distribution between dot and box joins as well as the predominant and typical use criteria values of these operators. We present the common range for the use and depth of the join operators to gauge the variations in the these values.

Measure	**PU** (Mode)	**TU** (Median)	**Percentage Distribution**
Dot Join	6*	15	90.6% } = 100%
Box Join	1*	0	9.4% }
Depth of Joins	2	2	-

Table 6.7: Dot and Box Joins in Alloy Models

Results: Table 6.7 shows the predominant and typical use values for the join operators and the depth of joins along with the percentage distribution between the dot and box join operators. The dot join operator is clearly more prevalent in Alloy models compared to the box join operator and accounts for 90.6% of all join operators whereas the box join operator constitutes only 9.4% of join uses. A typical Alloy model contains 15 uses of the dot join operator but no use of the box join operator. Joins in Alloy are fairly shallow as shown by the predominant and typical use values for the depth of joins that are both two.

[1] `(x,y) in f1.f2 <=> some z | (x,z) in f1 and (z,y) in f2`

Measure	12.5^{th}	25^{th}	50^{th}	75^{th}	87.5^{th}	Common Range
Dot Join	0	2	15	54	108	$[0, 108]$
Box Join	0	0	0	3	9	$[0, 9]$
Depth of Joins	2	2	2	3	3	$[2, 3]$

Table 6.8: Percentile Distribution of Dot and Box Join Uses and Depth of Joins

Table 6.8 shows the percentile distribution and common range of the uses of join operators and the depth of joins. There is a noticeable amount of variation in the frequency of dot join given that the common range is $[0, 108]$. We find a similar trend in the frequency of box joins. The common range for the depth of joins is $[2, 3]$ which implies that 75% of all join expressions bind two or three relations. We come to the conclusion that while the use of join operators (especially dot joins) is a common occurrence in Alloy models, deep joins are fairly uncommon.

Findings: Based on the results of this research question, we conclude that:

- The dot join operator is used extensively in Alloy models but the depth of joins is fairly shallow.

RQ# 39: What is the typical depth of quantification in Alloy models?

Motivation: Constraints in Alloy can be quantified using a number of built-in quantifiers. A quantified constraint takes the form `Q a: B | F` where `Q` is a quantifier, `x` is the quantified variable of kind `B` and `F` is a formula denoting the constraint. There are six quantifiers in the Alloy language:

- `all a: B | F`: the constraint `F` holds for *every* element `a` in `B`
- `some a: B | F`: the constraint `F` holds for *some* element `a` in `B`
- `no a: B | F`: the constraint `F` holds for *no* element `a` in `B`
- `lone a: B | F`: the constraint `F` holds for *at most one* element `a` in `B`
- `one a: B | F`: the constraint `F` holds for *exactly one* element `a` in `B`

Alloy allows for nested quantification in expressions *e.g.,* the constraint `all a: B | lone a.f` states that every element `a` in `B` is mapped to at most one element in the field `f`.

Nested quantifiers are generally believed to affect analysis complexity and solving time. We explore the depth of quantification to find if Alloy modelers are making use of deep quantified constraints that can be the target of future optimizations.

Approach: We extract from the model top-level quantified formulas and compute the depth of quantification by examining the nesting of quantification in these top-level formulas. We report the predominant and typical use criteria values for the depth of quantification.

Measure	**Predominant Use** (Mode)	**Typical Use** (Median)
Depth of Quantification	1	1

Table 6.9: Depth of Quantification in Alloy Models

Measure	**12.5th**	**25th**	**50th**	**75th**	**87.5th**
Depth of Quantification	1	1	1	1	2

Common Range: $[1, 2]$

Table 6.10: Percentile Distribution of Depth of Quantification

Results: Table 6.9 shows the results of this research question. We find that the predominant and typical depth of quantification are both one. The common range shown in Table 6.10 shows that the vast majority of quantification depths are 1 with a smaller number being two. We conclude that nesting quantifiers is a fairly uncommon practice in Alloy.

Findings: We find that:

- The depth of quantification in Alloy formulas is shallow. Optimizations that target deeply quantified constraints may not yield substantial improvements for the solving time in Alloy models.

6.4 Field Arity

High-arity fields are often discouraged in Alloy models. This section explores the arity of fields declared under signatures to determine if modelers are creating fields with high arity that may have an impact on analysis complexity and solving time.

RQ# 40: Are high-arity fields common in Alloy models?

Motivation: Fields in Alloy are declared in the body of signatures. The following signature declaration `sig A {f: e}` contains a field declaration `f: e` whose domain is the signature under which it is declared *i.e.,* `A` and whose range is given by the expression `e` that may include one or more sets. For example, the signature declaration `sig A {f1: B -> lone C}` instantiates a new field `f1` with arity three. Note that the smallest field arity is two since a field must bind the signature under which it is declared with at least one other set. We explore field arities in Alloy models to determine if modelers are frequently creating high-arity fields that could be addressed in future optimizations.

Approach: We extract from the model all field declarations and compute their arities by counting the number of sets in the type expression of the field declaration plus one for the signature under which the field is declared. We report the predominant and typical use values as well as the common range for field arities in our corpus of models.

Measure	**Predominant Use** (Mode)	**Typical Use** (Median)
Field Arity	2	2

Table 6.11: Field Arities in Alloy Models

Measure	**12.5$^{\text{th}}$**	**25$^{\text{th}}$**	**50$^{\text{th}}$**	**75$^{\text{th}}$**	**87.5$^{\text{th}}$**
Field Arity	2	2	2	2	3

Common Range: $[2, 3]$

Table 6.12: Percentile Distribution of Field Arities

Results: The results of this research question are presented in Tables 6.11 and 6.12. The predominant and typical use values for field arity are both two, which indicates that the most-frequent and typical field arities in Alloy are two (*i.e.,* the smallest field arity possible). The percentile distribution shown in Table 6.11 also showcases the prominence of fields with arity two given that 62.5% of all field arities are exactly two and 12.5% are exactly three. We conclude that high-arity fields are uncommon in Alloy models since the common range of field arities is $[2, 3]$.

Findings: We find that:

- High-arity fields are uncommon in Alloy models and thus are not a viable candidate for future optimizations.

6.5 Summary

In this chapter, we discussed a number of factors that are generally believed to impact analysis complexity and solving time. We examine the use of second-order operators (set cardinality and transitive closure) as well as the frequency of partial and total functions. We also explore the depth of joins and quantification in Alloy models. We find that second-order operators are prevalent in our corpus of models. Modelers are also using a considerable number of total and partial functions. The use of the dot join operator is widespread in Alloy models, but the depth of joins is generally shallow. Similarly, nesting quantifiers is an uncommon practice for Alloy modelers. Lastly, we examine field arities in Alloy models to determine the frequency of high-arity fields. The findings in this chapter are all aimed at optimization developers and are presented below.

Findings for Optimization Developers:

- Developers are encouraged to create solvers and optimizations that address the abundant use of the set cardinality operator in Alloy.
- Given that 26.8% of models in our corpus contain transitive closure operators and given the relatively substantial number of uses of transitive closure operators in these models, we encourage developers to explore developing optimizations centered around these operators.
- Total functions are used extensively in Alloy models. Thus, it is worthwhile for developers to work on improving the analysis methods with a focus on total functions.
- Partial functions are used modestly in Alloy models. Developers could potentially create optimizations that handle these functions.
- The dot join operator is used extensively in Alloy models but the depth of joins is fairly shallow.
- The depth of quantification in Alloy is shallow. Optimizations that target deep quantified constraints would probably not be effective on Alloy models.
- High-arity fields are uncommon in Alloy models and thus are not a viable candidate for future optimizations.

Chapter 7

Related Work

To the best of our knowledge, our Alloy corpus study is the first of its kind to study to explore the modeling characteristics of Alloy users. Other Alloy corpus studies examine the semantics of the models and focus on optimizations for the language or the development of external tools. Previous studies briefly discuss the syntactic features and constructs of the Alloy language but do not provide an in-depth look at the complex patterns in Alloy models beyond a simple tallying up of constructs. We briefly examine other corpus studies performed on programs written in object-oriented programming languages. We also discuss the object-oriented literature that inspired some of the research questions in this work. Finally, we provide a brief overview of two profiling techniques that we attempted to use for this work.

7.1 Alloy Profiling

Wang *et al.* [56] correlate Alloy model features with analysis time. They examine a number of static features of Alloy models at three different levels: an Alloy model, its Kodkod model, and its SAT model. The tally of these features on 119 Alloy models (103 from the Alloy Analyzer release 4.2) plus analysis of these models at varying scopes is used to train a machine learning component to predicate the best SAT solver from the characteristics of the problem. They found the features extracted from the Kodkod model to be the most valuable for predicting the best solver with the lowest analysis time. In comparison with our work, the 123 features Wang *et al.* extracted at the Alloy model level are based on counting the number of occurrences of types of nodes (*e.g.,* particular operators) in the

abstract syntax tree (AST), and include properties of the entire AST such as how many nodes are in the tree.

Erata [29] profiled some syntactic characteristics (*e.g.,* arity, use of transitive closure, use of integers) of a set of 109 Alloy models for discussion on the Alloy mailing list. Compared to these efforts, our work is more ambitious given that it aims to understand how people write Alloy models (rather than solver performance). We look at a more general set of Alloy models (2,138 models) and draw conclusions from larger patterns within the model's syntax. Unlike our corpus study, Erata's work does not provide any recommendations or actionable items based on the findings.

Ringert and Wali [48] performed a semantic pairwise comparison of Alloy models. The developed method converts two Alloy models into a single model that can generate instances of each model. The proposed algorithm was integrated into the existing Alloy code base. The comparison can thus be conducted in the Alloy Analyzer without the need to use external tools. The developed tool can check if two models are equivalent or if a model is a refinement or extension of another model. The corpus of models used to conduct an evaluation of the proposed method contained 654 Alloy models gathered from different sources including the iAlloy and Platinum evaluation models (both of which are included in our corpus). The translation algorithm has some limitations but it is still effective on the vast majority of models in the collected corpus. The researchers found that their proposed method yielded favorable results in terms of effectiveness and analysis cost. Ringert and Wali performed some rudimentary profiling of the models in their corpus and reported the sizes of Alloy models in terms of lines of code and numbers of signatures, fields, facts, and functions/predicates (including those from imported models). Unlike our work, Ringert and Wali examined models from a semantic point of view. Their work does not attempt to examine how modelers use the syntactic features of Alloy and does not attempt to identify particular patterns or trends in the models.

7.2 Software Metrics and Corpus Studies

Chowdhury and Zulkernine [26] use complexity, cohesion and coupling metrics as a means to predicate software vulnerability in programs. The authors provide empirical evidence that non-cohesive programs that exhibit high degrees of complexity and coupling are generally less secure. Chowdhury and Zulkerine use a suite of complexity, cohesion and coupling (CCC) metrics often used to assess the quality of software programs over their development life cycle. The CCC metrics include Source Line of Code (SLOC), Lack of Cohesion in Methods (LCOM), Number of Children (NOC), Depth and Width of Inheritance Trees

(DIT and WIT), *etc.* We have drawn inspiration from Chowdhury and Zulkernine's work when formulating some of our research questions (*e.g.,* length of Alloy models) and metrics (*e.g.,* SCG, Extension/Subset Hierarchy Graphs). Nevertheless, our work does not aim to assess the quality of Alloy models or uncover safety vulnerabilities in them. Chowdhury and Zulkerine conducted a case study using bug reports from fifty-two Mozilla Firefox releases to validate their framework.

We have drawn inspiration from object-oriented programming and UML modeling metrics such as those described in Briand and Jüst [25] and SDMetrics [58]. Briand and Jüst performed a literature review of 99 studies that measure four software quality attributes: reliability, maintainability, effectiveness and functionality. The study aimed to identify links between object-oriented measures and quality attributes. The researchers found that complexity, cohesion, size and coupling measures are more closely related to reliability and maintainability than inheritance measures. However, inheritance measures can still affect reliability and maintainability depending on the context and structure of the program.

Lopes and Ossher [43] conducted a quantitative study on a large corpus of 30,911 Java projects of different sizes using linear regression and found that certain characteristics of programs differ significantly with program size. Lopes and Ossher also found that the internal structure of programs can vary depending on the scale of the project and thus scale and program size have significant implications on object-oriented software metrics that do not currently account for differences in program size and scale. Their findings reinforce the idea that programming-in-the-small differs considerably from programming-in-the-large. Lopes and Ossher propose corrective measures that can be applied to existing software metrics to ensure that these metrics are size-independent. Our work takes inspiration from Lopes and Ossher's study given that we correlate certain model characteristics with features such as the number of signatures, predicates and functions. We also correlate certain Alloy model features using linear regression. However, our findings do not have any implications on metrics given that no previous studies have devised such metrics for Alloy models.

7.3 Profiling Techniques

When we first set out to devise a static analysis methodology, we explored different tools used in the industry. SonarQube [16] is an open-source platform used for code inspection and static analysis of code to detect bugs and security vulnerabilities in more than twenty programming languages. Support for additional languages can be added to SonarQube. For instance, Ruiz *et al.* [49] developed a tool called Xtext2Sonar capable of taking a

domain-specific language grammar written in Xtext [17] and generate a Java plugin for the SonarQube platform. The plugin allows developers to perform code quality checks on any file written in the domain-specific language. We did not make use of SonarQube and Xtext2Sonar due to SonarQube's quality-oriented set of features that aim to identify potential problems with the code as opposed to providing profiling statistics.

Rascal [12] is a metaprogramming language used to construct parsers, analyze and transform source code, and define new domain-specific languages. We considered using Rascal to perform our static analysis profiling, but adding support for Alloy in Rascal proved to be more time-consuming than developing our own ANTLR parser because Rascal does not provide automatic parser generation and thus adding support for Alloy in Rascal would have required a considerable amount of implementation work on our end.

Chapter 8

Conclusion

This thesis presents a methodology to profile Alloy models and identify patterns in them. We build an ANTLR parser for Alloy capable of generating a parse tree from a syntactically sound model. We use the XPath querying language in tandem with parse tree matching to extract patterns from the parse trees of models. We devise a number of research questions that differ in purpose and complexity. Research questions in the "Characteristics of Models" category aim to assess the "surface-level" features of models whereas the "Patterns of Use" questions use complex patterns to investigate the use of the Alloy language constructs. The "Analysis Complexity" questions examine the use of model features and constructs that may impact analysis complexity and solving time. For each research question, we report one or more data summary criteria including the predominant use (mode), typical use (median), percentage distribution, percentile distribution and common range. The data summary criteria are carefully chosen to accurately answer the research question and account for the skewness of the generated data.

When examining the characteristics of models, we explore the length of Alloy models as well as the use of signatures in terms of number and kind (top-level, subsignature extensions, subset signatures, *etc.*). We also explore the formula count in Alloy models and the use of constraint containers (predicates, functions and assertions) and command queries. Lastly, we discuss the corner cases of the Alloy language by assessing the use of five uncommon features: union supersets, bit shifting operators, set constants, macros and fields defined using set union and set difference. We use linear regression as a statistical tool to determine the best predictors for model length and field count. Our results show that there exists a high positive correlation between the number of formulas and the model length and between the number of top-level signatures and the field count.

We explore various patterns of use in Alloy to get a better understanding of how modelers are using the language's constructs. We investigate the use of library and user-created modules as well as integers. We also examine the use of sets (signatures and fields) in Alloy and the hierarchies that result from the relationships between them. We devise external structures that depict the relationships that exist among sets. Signature Hierarchy Graphs showcase the hierarchy that results from the subsignature extensions and subset signatures whereas the Signature Connectedness Graph reflects the degree of connectedness among the signatures of a model. Additionally, we investigate the use of the different formula modeling styles and attempt to identify how modelers are setting scopes in their models.

The "Analysis Complexity" research questions discuss the use of Alloy model features and constructs that are generally believed to impact solving time. We explore the use of second-order operators (set cardinality and transitive closure) as well as total and partial functions. We also assess the use of the dot and box join operators along with the depth of joins. We examine nested quantified expressions and assess the depth of quantification in Alloy models. Lastly, we examine field arity to determine if modelers are creating high-arity fields that may impact affect analysis complexity.

8.1 Findings

From the results of our research questions, we formulate findings divided into three categories: findings for language and tool designers, findings for educators and findings for optimization developers. Findings for language and tool designers aim to guide the evolution of the Alloy language by adding new constructs and features or removing rarely used ones. We present educators with a series of findings to help them identify which constructs and practices to emphasize when teaching student modelers. The findings aimed at optimization developers provide suggestions for back-end improvements. We present a summary of our findings:

Findings for Language Designers:

- Language designers are encourage to explore type checking mechanisms for the Alloy language to provide faster feedback to users because types can be used to partition a universe of atoms.
- The abundant use of scalars in Alloy models is evident and may warrant attention

from language and tool designers, who should consider adding syntactic sugar that allows modelers to create scalars directly.

- Language designers may want to remove the ability to `run` functions from the Alloy language given the scarcity of its use.
- Language designer may want to consider removing assertions from future versions of Alloy since they are a redundant construct that is not used frequently.
- Language designers may want to consider removing the ability to name constraint blocks in `run` and `check` commands since it is not widely used and it does not offer any advantages in terms of reusing the constraints.
- Language designers should consider dropping support for bit shifting operators in future versions Alloy so that solving engines do not have to support these operations.
- Given the scarcity of fields declared using set union and set difference, language designers may want to remove support for these expression in fields to simplify the language.
- Since user-created modules are used frequently in Alloy, language and tool designers may want to develop model management IDEs to help modelers manage these imported user-created modules.
- Some library modules are rarely used and hence language designers may want to drop them in future versions of Alloy.
- Integers in Alloy take more time in analysis. Language designers should add built-in identifiers that model numeric constants (*e.g.,* `One`, `Two`, *etc.*) to satisfy the majority of integer uses without actually using integers for analysis in addition to modifying the Alloy Analyzer to allow it to warn users about integer fields that are only used as a linear order.
- Modelers are using the set cardinality operator to set the size of signature sets. Given that this practice is inefficient in analysis, tool designers are encouraged to add a warning that discourages its use and provides alternative methods to specify the signature set size (multiplicity keywords, command queries).
- Language and tool designers may want to give modelers access to the Signature Connectedness Graph (SCG) to allow them to gauge the connectedness of their models.

- A dedicated structure/record construct is not needed in Alloy.
- Abstract signatures are almost always extended with subsignature extensions.
- The ordering module is commonly applied to sets with a non-exact scopes. We recommend that tool designers discourage this practice by warning modelers when they use a non-exact scope with an ordered set because it can give non-intuitive results.

Findings for Educators:

- Educators are encouraged to highlight the value of abstract signatures because they allow modelers to take advantage of inheritance and scope inference in addition to making the models more concise and easier to modify in the future (the union of multiple signatures can be replaced with a reference to the parent abstract signature).
- Given that `enum`s are an underutilized construct in Alloy, educators are encouraged to highlight the use of `enum`s to concisely instantiate multiple ordered signatures with multiplicity `one`.
- Educators are encouraged to ensure that student modelers are using signature facts correctly to avoid erroneous results. Alternatively, educators may want to discourage the use of signature facts.
- Educators should encourage student modelers to use unparameterized predicates instead of assertions to simplify the language.
- Given their untyped nature, macros are significantly more flexible than predicates and functions and can be used in ways that other constraint holders cannot. Since macros are underutilized in Alloy models, educators are encouraged to highlight them and promote their use when teaching student modelers.
- Modelers are not using the ordering module to its full extent given the profuseness of integers used solely to impose a linear ordering. Educators are encouraged to promote the use of the ordering module.
- We encourage educators to highlight the importance of using multiplicity keywords and command queries to set the size of signature sets in lieu of the set cardinality operator because setting the size of a signature using the set cardinality operator leads to longer solving times.

- Educators may want to consider the frequency of use of the different formula modeling styles when deciding which modeling style to teach first or emphasize.
- Modelers tend to use non-exact scopes far more frequently than exact scopes. Thus, educators are encouraged to explore the different scope categories with student modelers and highlight the performance gains that can be achieved by using exact scopes.
- Educators should discourage the use of non-exact scopes with an ordered set.
- Integer scopes should be explained in greater detail. Educators should also encourage students to find alternative ways of modeling problems with numeric constants that do not involve integers.

Findings for Optimization Developers:

- Given that union supersets are scarce in Alloy models, optimization developers can safely omit support for union supersets when developing their optimizations.
- Developers can safely omit support for fields declared using set union and set difference in their optimizations without significantly compromising the usefulness of the tools.
- Developers are encouraged to create optimizations for the ordering module and for integers in Alloy because they are used frequently.
- Optimization developers are encouraged to internally transform integer applications used to impose a linear order to ordered sets that take advantage of the ordering module because the use of integers in Alloy models is discouraged due to its impact on solving time.
- We recommend that developers invest in the development of Alloy optimizations that convert applications of the set cardinality operators that specify the size of a signature set to command queries or signature declarations with a multiplicity keyword because specifying the size of a signature using the set cardinality increases model complexity and solving time.
- Given the fairly shallow depth of extension hierarchy graphs in Alloy models, optimizations do not necessarily need to scale favorably for deep extension hierarchies.
- Creating subsets of a subset is a fairly rare practice in Alloy. Hence, optimization developers do not necessarily need to account for this particular instance of set hierarchy when developing their tools and engines.

- The shallow signature hierarchies in Alloy models are another argument in favor of a type system in the Alloy language.
- Developers are encouraged to create solvers and optimizations that address the abundant use of the set cardinality operator in Alloy.
- Given that 26.8% of models in our corpus contain transitive closure operators and given the relatively substantial number of uses of transitive closure operators in these models, we encourage developers to explore developing optimizations centered around these operators.
- Total functions are used extensively in Alloy models. Thus, it is worthwhile for developers to work on improving the analysis methods with a focus on total functions.
- Partial functions are used modestly in Alloy models. Developers could potentially create optimizations that handle these functions.
- The dot join operator is used extensively in Alloy models but the depth of joins is fairly shallow.
- The depth of quantification in Alloy formulas is shallow. Optimizations that target deeply quantified constraints would probably not be effective on Alloy models.
- High-arity fields are uncommon in Alloy models and thus are not a viable candidate for future optimizations.

8.2 Threats to Validity

Our results in evaluating a set of research questions on a corpus of Alloy models allows us to make claims regarding common characteristics and patterns in Alloy modeling. In this section, we consider the threats to the validity of our results.

External Validity: The results of this study were derived by examining a corpus of scraped Alloy models in addition to a number of models provided by Jackson's book on Alloy and previous studies. While the randomized nature of the model selection process improves the generality and applicability of the results, we cannot ignore the possibility of obtaining different results when running our scripts on another corpus of models. We note that we do not classify or categorize the models to differentiate between expert and novice modelers. Replicating this study on a corpus of models that pertains to one particular

category of modelers could produce results that deviate from the ones presented in this work. We also do not categorize models by size or date. By aggregating the analyses over the whole corpus, the results presented in this work may mask the fact the results might be different for certain subgroups of the corpus (*e.g.,* large vs. small, old vs. new, complete vs. incomplete, *etc.*).

Internal Validity: A purely syntactic static analysis of Alloy models is used to derive our results. We acknowledge that incorrect expressions, commented-out text, and unfinished models could skew our results. Our analysis is performed on a per-file basis and thus our profiling may have missed certain characteristics and patterns (*e.g.,* inheritance, pure/dominant formula modeling style, use of certain constructs, *etc.*) in models that span over multiple files. We do take some cautionary measures to address these shortcomings in our profiling. The measures are explained as need in the research questions discussed in Chapters 4, 5 and 6. Our parser was tested extensively to ensure that it can properly parse any syntactically correct model written Alloy versions 3-5. The individual scripts used to answer the research questions were thoroughly tested with a number of unit tests to ensure that all variations of a particular query can be detected and extracted successfully. For some research questions, we present the non-zero data summary criteria since the all-inclusive values are all zeros and do not provide any meaningful insights. We acknowledge that the non-zero data summary criteria may present an inflated use frequency for some constructs.

Construct Validity: Some research questions in this work are inspired by concepts used to assess object-oriented programs (*e.g.,* set hierarchy graphs, signature connectedness graph (SCG), *etc.*). Alloy is a modeling language and differs significantly from programming languages, which may affect the applicability and relevance of these research questions. We also acknowledge that other research questions could be devised to assess different aspects of Alloy models. The observations and inferences made in this study could change significantly if additional measures or Alloy constructs are considered for each research question.

8.3 Future Work

Our corpus study is the first of its kind to perform a static analysis of Alloy models. While our corpus contains a large number of diverse Alloy models, it does not distinguish between the different kinds of models. Future studies may want to disaggregate the corpus of models to get a better understanding of the habits and trends of different Alloy modelers. For instance, models used an educational environment may exhibit different characteristics and

patterns than the ones used in industry. A categorization of models based on size (*i.e.,* small vs. large) could yield a number of insights into the differences that separate smaller Alloy models from larger ones. We believe that our study presents a good assessment of the current state of Alloy modeling. However, a disaggregation of models by timestamps or date (*i.e.,* old vs. new) could help us determine if the use of language features and modeling idioms have changed over time.

This thesis lays the foundation for a number of future research directions. Not only can the findings aimed at educators be used to help student modelers but they can also be incorporated into future literature on the Alloy language to provide stylistic and functional guidelines for new modelers. The findings aimed at language and tool designers can help the evolution of the Alloy language and the Alloy Analyzer by adding new constructs or removing unused ones and providing modelers with additional features and warnings in the Analyzer. Developers can also benefit from the findings presented in this work to help them identify key components and practices in the language that can be the target of future optimizations.

In Chapter 6, we assessed the use of Alloy model features and constructs that are believed to affect analysis complexity and solving time. However, we did not prove the existence of any correlation between solving time and model features. Future research can attempt to correlate solving time with these model features and constructs to corroborate these claims. A linear regression could be conducted as a statistical means of identifying any correlations that exist between solving time and model features. In particular, multiple linear regression (MLR), which is a statistical technique that uses several explanatory variables to predict the outcome of a response variable, could be used to correlated multiple model features with solving time. Our work found that usage of set cardinality, transitive closure and dot joins is relatively frequent in Alloy models. Future work can attempt to correlate these operators with solving time.

References

[1] ABZ 2021 – 8th International Conference on Rigorous State Based Methods. https://abz2021.uni-ulm.de. Last accessed: 2021-02-03.

[2] CUP User's Manual. https://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html. Last accessed: 2021-07-22.

[3] A Guide to Alloy - Dynamics II. https://www.doc.ic.ac.uk/project/examples/2007/271j/suprema_on_alloy/Web/tutorial2_2.php. Last accessed: 2021-06-26.

[4] org.alloytools.alloy. https://alloytools.org. Last accessed: 2021-02-03.

[5] Questions tagged [alloy]. https://stackoverflow.com/tags/alloy. Last accessed: 2021-02-03.

[6] The R Project for Statistical Computing. https://www.r-project.org. Last accessed: 2021-02-03.

[7] Welcome to Alloytools. https://alloytools.discourse.group. Last accessed: 2021-02-03.

[8] What does it mean to "run" a function in Alloy? https://stackoverflow.com/questions/68146285/what-does-it-mean-to-run-a-function-in-alloy. Last accessed: 2021-07-15.

[9] What is the difference between assertions and unparameterized predicates in Alloy? https://stackoverflow.com/questions/68076177/what-is-the-difference-between-assertions-and-unparameterized-predicates-in-allo. Last accessed: 2021-06-26.

[10] What is the purpose of abstract signatures with no fields in Alloy? https://stackoverflow.com/questions/67992060/what-is-the-purpose-of-abstract-signatures-with-no-fields-in-alloy. Last accessed: 2021-07-21.

[11] The util/ordering module and ordered subsignatures. https://stackoverflow.com/questions/17308778/the-util-ordering-module-and-ordered-subsignatures, 2013. Last accessed: 2021-06-02.

[12] Rascal. https://www.rascal-mpl.org/, 2014. Last accessed: 2021-06-05.

[13] 5.0.0 change list. https://github.com/AlloyTools/org.alloytools.alloy/wiki/5.0.0-Change-List, 2018. Last accessed: 2021-02-07.

[14] Catalyst. https://github.com/WatForm/alloy-model-sets, 2021. Last accessed: 2021-05-26.

[15] Logic for Systems, Computer Science, Brown University. http://cs.brown.edu/courses/cs171/, 2021. Last accessed: 2021-02-03.

[16] SonarQube. https://www.sonarqube.org/, 2021. Last accessed: 2021-06-05.

[17] Xtext. https://www.eclipse.org/Xtext/, 2021. Last accessed: 2021-06-05.

[18] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[19] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[20] Amin Bandali. *A Comprehensive Study of Declarative Modelling Languages*. MMath thesis, University of Waterloo, Cheriton School of Computer Science, 2020.

[21] Thomas Ball and Benjamin Zorn. Teach foundational language principles. *Communications of the ACM*, 58(5):30–31, May 2015.

[22] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[23] Jean-Louis Boulanger, editor. *Formal Methods Applied to Industrial Complex Systems*. Wiley, 2014.

[24] Jonathan P Bowen. Z: A formal specification notation. In *Software Specification Methods*, Formal Approaches to Computing and Information Technology FACIT, pages 3–19. Springer, London.

[25] Lionel C. Briand and Jürgen Wüst. Empirical studies of quality models in object-oriented systems. volume 56 of *Advances in Computers*, pages 97–166. Elsevier, 2002.

[26] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of systems architecture*, 57(3):294–313, 2011.

[27] Diego de Azevedo Oliveira and Marc Frappier. Verifying SGAC Access Control Policies: A Comparison of ProB, Alloy and Z3. In Alexander Raschke, Dominique Méry, and Frank Houdek, editors, *Rigorous State-Based Methods*, pages 223–229. Springer International Publishing, 2020.

[28] Aboubakr Achraf El Ghazi and Mana Taghdiri. Relational Reasoning via SMT Solving. In *FM 2011: Formal Methods*, Lecture Notes in Computer Science, pages 133–148, Berlin, Heidelberg. Springer Berlin Heidelberg.

[29] Ferhat Erata. Alloy/kodkod benchmarks. https://tinyurl.com/alloy-benchmarks, 2018. Last accessed: 16 Feb 2021.

[30] Sabria Farheen. *Improvements to Transitive-Closure-based Model Checking in Alloy*. MMath thesis, University of Waterloo, Cheriton School of Computer Science, 2018.

[31] Gomaa Hassan. *Software modeling and design : UML, use cases, patterns, and software architectures*. Cambridge University Press, Cambridge, 2011.

[32] Brian Henderson-Sellers, Larry L. Constantine, and Ian M. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158, 1996.

[33] Dennis E Hinkle, William Wiersma, and Stephen G Jurs. *Applied statistics for the behavioral sciences*, volume 663. Houghton Mifflin College Division, 2003.

[34] Tamjid Hossain and Nancy A. Day. Dash+: Extending Alloy with Hierarchical States and Replicated Processes for Modelling Transition Systems. *to appear in International Workshop on Model-Driven Requirements Engineering (MoDRE) @ IEEE International Requirements Engineering Conference (RE)*, 2021.

[35] Nghi Huynh, Marc Frappier, Herman Pooda, Amel Mammar, and Régine Laleau. SGAC: A multi-layered access control model with conflict resolution strategy. *The Computer Journal*, 62(12):1707–1733, 2019.

[36] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM TOSEM*, 11(2):256–290, 2002.

[37] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, Revised edition, 2012.

[38] Daniel Jackson. Alloy: A language and tool for exploring software designs. *Communications of the ACM*, 62(9):66–76, September 2019.

[39] S. C. Johnson. Lint, a c program checker. 1978.

[40] Clifford B. Jones. *Systematic software development using VDM (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[41] Khadija Tariq. *Linking Alloy with SMT-based Finite Model Finding*. MMath thesis, University of Waterloo, Cheriton School of Computer Science, 2021.

[42] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[43] Cristina V Lopes and Joel Ossher. How scale affects structure in java programs. *SIGPLAN notices*, 50(10):675–694, 2015.

[44] Shin Nakajima. Using Alloy in Introductory Courses of Formal Methods. In *Structured Object-Oriented Formal Language and Method*, pages 97–110. Springer International Publishing, 2015.

[45] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, 2015.

[46] James Noble, David J. Pearce, and Lindsay Groves. Introducing Alloy in a Software Modelling Course. In *Formal Methods in Computer Science Education Workshop*, 2008.

[47] Terrance Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[48] Jan Ringert and Syed Wali. Semantic Comparisons of Alloy Models. ACM / IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS), 10 2020.

[49] Iván Ruiz-Rube, Tatiana Person, Juan Manuel Dodero, José Miguel Mota, and Javier Merchán Sánchez-Jara. Applying static code analysis for domain-specific languages. *Software and systems modeling*, 19(1):95–110, 2020.

[50] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[51] Anjali Sree-Kumar, Elena Planas, and Robert Clarisó. Analysis of feature models using alloy: A survey. *Electronic Proceedings in Theoretical Computer Science*, 206:46–60, 03 2016.

[52] Allison Sullivan, Kaiyuan Wang, Sarfraz Khurshid, and Darko Marinov. Evaluating state modeling techniques in alloy. In *Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications*, 2017.

[53] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 632–647. Springer Berlin Heidelberg, Berlin, Heidelberg.

[54] Amirhossein Vakili and Nancy A Day. Finite Model Finding Using the Logic of Equality with Uninterpreted Functions. In *International Symposium on Formal Methods*, pages 677–693. Springer, 2016.

[55] World Wide Web Consortium (W3C). Xml path language (xpath) 3.1. https://www.w3.org/TR/2017/REC-xpath-31-20170321/. Last accessed: 2021-02-03.

[56] W. Wang, K. Wang, M. Zhang, and S. Khurshid. Learning to Optimize the Alloy Analyzer. In *ICST*, pages 228–239. 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), 2019.

[57] Hillel Wayne. *Practical TLA+: Planning Driven Development*. Apress, 2018.

[58] Jürgen Wüst. Sdmetrics. https://www.sdmetrics.com/. Last accessed: 16 Feb 2021.

[59] Pamela Zave. Using Lightweight Modeling to Understand Chord. *ACM SIGCOMM Computer Communication Review*, 42(2):49–57, 2012.

[60] Guolong Zheng, Hamid Bagheri, Gregg Rothermel, and Jianghao Wang. Platinum: Reusing constraint solutions in bounded analysis of relational logic. In *Fundamental Approaches to Software Engineering*, pages 29–52, Cham, 2020. Springer International Publishing.

APPENDICES

Appendix A

Alloy Language Grammar

A.1 ANTLR Notation

The grammar presented in this appendix uses the ANTLR standard notion and contains the following operators:

- `:` for starting a rule definition;
- `;` for ending a rule definition;
- `'x'` for a char or string literal `x`;
- `x*` for zero or more repetitions of `x`;
- `x+` for one or more repetitions of `x`;
- `x|y` for a choice of `x` or `y`;
- `x?` for an optional `x`

in addition,

- `(x (',' x)*)?` means zero or more comma-separated occurrences of `x`
- `x (',' x)*` means one or more comma-separated occurrences of `x`

A.2 Grammar

The complete Alloy grammar used to generate the parser used in this work is presented below:

```
1  specification : module? open* module? paragraph* EOF | EOF;
2
3  module : 'module' name  ( '[' 'exactly'? name  (',' 'exactly'
      ? name)*  ']' )?;
4
5  names_opt: names?;
6
7  as_name_opt: ('as' name)?;
8
9  para_open: ('[' names_opt ']')?;
10
11 open : priv 'open' name para_open as_name_opt ;
12
13 macro_expr : '='? '{' (expr* | decls)'}' | '=' (expr | decls)
      ;
14
15 macro : 'let' name ('[' names ']')? macro_expr;
16
17 paragraph : factDecl | assertDecl | funDecl | cmdDecl |
      enumDecl | sigDecl | predDecl | macro;
18
19 factDecl : 'fact' name? block;
20
21 name_opt : name?;
22
23 assertDecl : 'assert' name_opt block;
24
25 nameID : name '.' ID | ID;
26
27 decls_f : '{' decls '}';
28
29 funExpr : expr;
30
31 funDecl : priv 'fun' nameID paraDecls_opt ':' expr '{'
      funExpr '}';
```

```
1  nameOrBlock: (name|block)? | (nameID block)?;
2
3  name_cmd_opt : (name ':')?;
4
5  scope_opt : scope?;
6
7  run_or_check : 'run'|'check';
8
9  cmdDecl : name_cmd_opt run_or_check nameOrBlock scope_opt;
10
11 paraDecls : '(' (decl (','decl)*)? ')' | '[' (decl (','decl)
      *)?']';
12
13 paraDecls_opt : (paraDecls)?;
14
15 predDecl : 'private'? 'pred' nameID paraDecls_opt block;
16
17 typescopes : typescope (',' typescope)*;
18
19 but_typescopes : ('but' typescopes)?;
20
21 expect_digit : ('expect' DIGIT)?;
22
23 scope : 'for' number but_typescopes expect_digit | 'for'
      typescopes expect_digit | 'expect' DIGIT;
24
25 exactly_opt: 'exactly'?;
26
27 typescope : exactly_opt number (name | 'seq' | 'int');
28
29 decls : (','? decl (','decl)* )?;
30
31 multiplicity: mult?;
32
33 abs : 'abstract'?;
34
35 priv : 'private'?;
36
37 abs_multiplicity : abs multiplicity | multiplicity abs;
```

```
1  sigDecl : priv abs_multiplicity 'sig' names sigExtension '{'
       decls '}' block_opt;
2
3  names: name (',' name)*;
4
5  enumDecl : 'enum' name '{' names '}';
6
7  mult : 'lone' | 'one' | 'some';
8
9  union: name ('+' name)*?;
10
11 superSet: name | union;
12
13 sigExt : 'extends' name | 'in' superSet ;
14
15 sigExtension: sigExt?;
16
17 exprs : (expr (',' expr)*)?;
18
19 notOp: ('!' | 'not')?;
20
21 decls_e : decl (',' decl)*;
22
23 expr : 'let' letDecl (',' letDecl)* blockOrBar
24     | quant decls_e blockOrBar
25     | unOp expr | expr binOp expr | expr arrowOp expr
26     | expr notOp compareOp expr
27     | expr ('=>'|'implies') expr 'else' expr
28     | 'sum' '['? exprs']'? | expr '[' exprs']' | disjFunc
29     | constant  |'int' | 'seq/Int' | '(' expr ')'| name | '@'
         name | 'this'
30     | block | quant expr binOp expr
31     | '{' decl (',' decl)* blockOrBar '}' | decls_f |STRING;
32
33 num : '-'? number;
34
35 const_sets: 'none' | 'univ' | 'iden';
36
37 constant : num | const_sets;
```

```
1  disjFunc : 'disj' '[' (expr (',' expr)*)? ']';
2
3  disjoint : ('disj' | 'disjoint')?;
4
5  disj : 'disj'?;
6
7  comma_opt : ','?;
8
9  decl : priv disjoint names ':' disj expr comma_opt | name '='
       expr;
10
11 letDecl : name '=' expr;
12
13 quant : 'all' | 'no' | 'some' | 'lone' | 'one' | 'sum';
14
15 setCard: '#';
16
17 tcOp :  '*' | '^' ;
18
19 unOp : '!' | 'not' | 'no' | mult | 'set' | setCard | '~' |
     tcOp | 'seq';
20
21 bit_shifter_operators: '<<' | '>>' | '>>>';
22
23 dotOp: '.';
24
25 add: '+';
26
27 sub: '-';
28
29 binOp : '||' | 'or' | '&&' | 'and' | '<=>' | 'iff' | '=>' | '
     implies' | '&' | add | sub | '++' | '<:' | ':>' | dotOp |
     bit_shifter_operators;
30
31 mult_or_set : (mult |'set')?;
32
33 arrowOp : mult_or_set '->' mult_or_set;
34
35 rel_operators: '=' | '<' | '>' | '=<' | '>='| '<=';
```

```
compareOp : rel_operators | 'in';

block : '{' expr* '}';

block_opt: block?;

blockOrBar : block | bar expr;

bar : '|';

name : ('this/')? (ID '/')* ID;

DIGIT : [0-9] ;

number: DIGIT+;

ID : ALPHA ( ALPHA | DIGIT )* | STRING;

ALPHA : [a-zA-Z"\u0080-\uFFFF_]+;

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines

COMMENT : '/*' .*? '*/' -> skip;

LINE_COMMENT : ('//' | '--') ~[\r\n]* -> skip;

STRING : '"' (~'"'|'/"')* '"';
```