Static Profiling of Alloy Models

Elias Eid and Nancy A. Day

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. 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 1,652 Alloy models. To evaluate these research questions, we create a methodology that leverages the power of ANTLR pattern matching and the query language XPath. Our research questions are split into two categories depending on their purpose. The Model Characteristics category aims to identify *what* language constructs are used commonly. Modeling Practices questions are considerably more complex and identify *how* modelers are using Alloy's constructs. We also evaluate our research questions on a subset of models from our corpus written by expert modelers. We compare the results of the expert corpus to the results obtained from the general corpus to gain insight into how expert modelers use the Alloy language. We draw conclusions from the findings of our research questions and present actionable items for educators, language and environment designers, and tool developers. Actionable items for educators are intended to highlight underutilized language constructs and features, and help student modelers avoid discouraged practices. Actionable items aimed at language designers present ways to improve the Alloy language by adding constructs or removing unused ones based on trends identified in our corpus of models. The actionable items aimed at environment designers address features to facilitate model creation. Actionable items for tool developers provide suggestions for back-end optimizations.

Index Terms—Declarative modeling, Alloy, Static analysis

1 Introduction

Software modeling is becoming an important part of the software development process in order to manage complexity and reduce development effort. Formal declarative modeling languages, such as Alloy [1], TLA+ [2], B [3], Event-B [4], Z [5], VDM [6], and Abstract State Machines [7], are suitable for capturing structural and behavioral descriptions formally and abstractly in terms of sets, relations, and logical formulas. Automated search and proof-based techniques provide feedback regarding the correctness of the model early in the development process. Examples of the use of declarative modeling are: Zave's work using Alloy to discover problems in the Chord protocol [8]; Newcombe *et al.*'s work with TLA+ at Amazon [9]; and Huynh *et al.*'s work with B on describing a healthcare access control model [10].

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.,* [2], [3], [4], [5], [6], [11], [12]) and there are conferences dedicated to the paradigm (*e.g.,* the ABZ conference series [13]). There are also compilations of case studies or comparisons of modeling practices using these and related languages (*e.g.,* [14], [15], [16], [17]) and university courses that teach some of these languages (*e.g.,* [18], [19], [20], [21]). Ball and Zorn [22] 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.

Our work aims to understand how people write Alloy models. In this article, we provide the first deep analysis of a general corpus of Alloy models (1,652 models) and a smaller expert corpus (75 models). We present a variety of research questions to investigate these models. We determined these research questions from 1) existing literature on measures in programming and modeling (*e.g.,* [23], [24], [25]); 2) Alloy teaching material and discussions (*e.g.,* Jackson's Alloy book [11], Alloy Discourse [26], Stack Overflow [27]), and 3) interactions with others in our research group investigating Alloy modeling and tools (*e.g.,* [28], [29], [30]).

We divide our research questions into two categories: 1) model characteristics; and 2) modeling practices. Model characteristics cover "surface-level" research questions that aim to identify *what* language constructs are used frequently. This includes the use of model features that may impact analysis complexity and solving time. Modeling practices research questions attempt to identify *how* the language constructs are used and consequently are significantly more involved than the questions in the model characteristics section. For each research question, we provide motivation, our approach to answering the question, a series of findings (stated in *italics*) that produce actionable items (prefixed with **Action item**) aimed at educators, language and environment designers, and tool developers.

Researching the characteristics and practices of models written in Alloy has a significant impact on pedagogy. As

- *Elias Eid (eeid@uwaterloo.ca) is with the David R. Cheriton School of Computer Science, University of Waterloo.*
- *Nancy A. Day (nday@uwaterloo.ca) is with the David R. Cheriton School of Computer Science, University of Waterloo.*

Manuscript received April 19, 2005; revised August 26, 2015.

declarative modeling becomes more popular and useful, it is important to examine how modelers use these languages in order to help educators promote good practices, acknowledge bad modeling practices and create teaching materials. Our work also benefits language designers who can use our results to add new features to the Alloy language. Designers of model development environments can also can use our findings to create new software applications to facilitate model creation. We refer to designers of model development environments as "environment designers" throughout this article. We also offer suggestions for tool developers who can incorporate back-end analysis optimizations that would be valuable because of common model features.

2 Background: Alloy

Alloy is a modeling language that can express the fundamental structure and behavior of a system as constraints on sets and relations [1], [11], [31]. The Alloy language combines relational calculus and first-order logic with transitive closure and set cardinality operators and limited support for arithmetic. It contains a relatively small number of constructs making it an easy language to learn and analyze. The Alloy language is composed of the following constructs:

1) **Signature declarations:** introduce a new set of atoms. We use the term **signature** to refer to the unary set introduced by a signature declaration. Signatures may be declared with a multiplicity such as `one`, `some` and `lone` (*i.e.,* at most one) to specify the size of a signature. Signature declarations may contain fields. **Fields** are written in the body of signature declarations and are relations showing how signatures are connected to each other. Signature declarations may include a block known as a **signature fact block** containing formulas associated with the declared set.
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, or a function.
3) **Predicates:** are a group of formulas that can be referred to elsewhere in the model. Predicates may be parametrized, *i.e.,* they can include zero or more arguments.
4) **Functions:** are named expressions that return a value. Functions may be parametrized. Functions can be called in formulas.
5) **Assertions:** are a set of constraints that should follow from the facts of the model and can be checked in a command. Unlike predicates and functions, assertions do not take parameters.
6) **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 formulas. `check` commands search for a counterexample to a formula. `run` commands can be used with predicates and functions whereas `check` commands can be used with predicates, functions and assertions. Commands can be supplemented with **scopes** that limit the size of the signature sets in instances or counterexamples that will be considered.

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. The Alloy Analyzer [32] is used to edit and analyze Alloy models via its Kodkod engine [33] and SAT solvers. The Analyzer checks the queries of the model for finite sizes of the sets. We will explain the constructs of the Alloy language in more detail with examples as needed in the sections that follow.

3 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 [34] for scraping Alloy models from github repositories. The tool uses standard techniques to gather all publicly available Alloy models it can find, including the ones available with the Alloy Analyzer, ones scraped from public github repositories and other sources (*e.g.,* the 56 models provided in Jackson's book on Alloy [11]). We ensure there are no files that are exact duplicates of each other in this corpus, which includes removing 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 files that are part of the Alloy language/Analyzer[1]. We exclude files that do not parse correctly with the Alloy Analyzer version 5, 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. We follow the import statements (excluding those for library imports) to determine the user-created Alloy model from a set of files. In total, our corpus contains 1,652 Alloy models (containing 1,845 files), which includes models drawn from 504 different github repositories. Within these, there are 31 models created by our research group prior to this work. We refer to this corpus of models as the "general corpus".

We identify a subset in our corpus consisting of Alloy models written by expert modelers. These hand-selected models were either used in industry or in peer-reviewed publications to model real-world complex systems. The subset of expert models consists of 75 Alloy models (containing 92 files) drawn from 10 different github repositories. We use the term "expert corpus" to refer to the subset of expert models. A comprehensive list of these expert models and their sources can be found in [35].

In our corpus, we assume that the 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. Our corpus may contain automatically generated models which means that one person's modeling style and preferences could be affecting many models.

1. Alloy library (or util) files have a distinct syntax.

4 Methodology

To answer our research questions, we statically analyze textual Alloy models. Each Alloy model is composed of one or more files. For each research question, we create a query, search for instances of the query in the models, and then collate the results across the models. For multi-file Alloy models, the result of a count query is the summation of the counts obtained from each file in the model. If a parametrized file is imported more than once (for different parameter values) it is included in our tallies multiple times.

To search for instances of a pattern in an Alloy 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. Originally, XPath was a query language for XML documents. Support for XPath was added to the parser-generator ANTLR [36] with its version 4. Thus, first we created and tested an ANTLR grammar for Alloy. Our ANTLR parser accepts models that are written in the input languages for the Alloy Analyzer versions 3 - 5[2]. The differences between these versions are very small. We examine the parse tree of the model (rather than the abstract syntax tree) because it contains all the information from the model including the whole string associated with each non-terminal and terminal, and it allows for seamless extraction of subtrees and nodes.

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. XPath can extract subtrees from the model's 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. An XPath hierarchy path is sufficient when the research question requires extracting one kind of subtree from the parse tree.

When a research question requires extracting a subset of a node kind or a subtree that conforms to a particular pattern that cannot be expressed in an XPath hierarchy path, we additionally use ANTLR's built-in parse tree matching. A **parse tree pattern** is a string that describes what we want to match in the model. It can contain terminals, non-terminals and strings from the grammar. Strings from the grammar correspond to the literal value of certain terminal rules.

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 model. 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 research questions. 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.

2. There are no syntactic changes in the language between versions 4 and 5 [37].

The data resulting from multiple Alloy models is combined using an R script [38]. 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 model identifies the most recurrent value in the collected values and thus identifies the most frequent form/use of each pattern.
- **Typical Use (TU):** The **median** per model 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 models 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 12.5th, 25th, 50th, 75th and 87.5th 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 12.5th percentile and the 87.5th 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. We report the results of each research for the general corpus and for the expert corpus separately. **The expert results shown in the tables are enclosed in parentheses.** In the following sections, we only discuss the results of the expert corpus in detail if they differ significantly from the ones obtained from the general corpus. The scripts used to perform our profiling of Alloy models are available on github [35].

5 Model Characteristics

In this section, we explore the "surface-level" characteristics of Alloy models. We present the data summary criteria for a number of Alloy language constructs and features and draw a series of findings and action items from the results.

RQ# 1: What are the characteristics of an Alloy model?
Motivation: The Alloy language contains a relatively small number of constructs. Certain constructs are used more frequently than others. By learning about the basic characteristics of Alloy models and the constructs most commonly used, educators can focus their attention on teaching more commonly-used constructs. Language and environment designers can determine where to concentrate their efforts on improving the language and its model creation software support. Tool developers can use these results to determine which constructs to target when creating optimizations.

TABLE 1
Model Characteristics
(An asterisk * indicates a non-zero criterion)

Measure	Predominant Use	Typical Use	Percentage Distribution	Common Range
Model Length	52 (73)	63 (76)	-	$[17, 204]$ $([26, 253])$
Model Span	1 (1)	1 (1)	-	$[1, 2]$ $([1, 2])$
Signature Count	2 (3)	8 (7)	-	$[2, 25]$ $([3, 28])$
Top-level Signatures	3 (3)	4 (3)	22.6% (32.5%) } = 100%	$[2, 9]$ $([2, 6])$
Subset Signatures	1* (1*)	0 (0)	1.5% (0.3%)	$[0, 0]$ $([0, 0])$
Subsignature Extensions	2* (4*)	3 (3)	73.9% (60.5%)	$[0, 15]$ $([0, 16])$
Signatures with Multiplicity `one`	1* (1*)	1 (1)	26.4% (41.5%) } of sigs	$[0, 10]$ $([0, 10])$
`enum`s	2* (4*)	0 (0)	1.9% (6.7%)	$[0, 0]$ $([0, 0])$
`abstract` Signatures	1* (1*)	1 (1)	10.3% (14.8%) of sigs	$[0, 4]$ $([0, 6])$
Fields	1 (1)	2 (2)	-	$[1, 4]$ $([1, 3])$
Signatures with Fields	-	-	17.8% (29.1%) } = 100%	-
Signatures without Fields	-	-	82.2% (70.9%)	-
Formulas	5* (10)	20 (34)	-	$[3, 93]$ $([6, 227])$
Signature Facts	1* (1*)	0 (0)	47.6% (9.5%) of sigs 41.0% (15.3%) of facts	$[0, 3]$ $([0, 3])$
Facts	1 (1)	2 (3)	-	$[0, 13]$ $([0, 11])$
Predicate Declarations	1 (1)	4 (5)	77.4% (63.1%) } = 100%	$[0, 13]$ $([1, 21])$
Function Declarations	1* (1*)	0 (1)	22.6% (36.9%)	$[0, 4]$ $([0, 4])$
Predicate Calls (as commands)	1* (1*)	1 (0)	8.4% (4.3%) } = 100%	$[0, 3]$ $([0, 3])$
Predicate Calls (within formulas)	4* (1*)	3 (0)	91.6% (95.7%)	$[0, 20]$ $([0, 52])$
Function Calls (as commands)	1* (1*)	0 (0)	0.1% (0.1%) } = 100%	$[0, 0]$ $([0, 0])$
Function Calls (within formulas)	2* (1*)	0 (1)	99.9% (99.9%)	$[0, 9]$ $([0, 9])$
Assertion Declarations	1* (1*)	2* (2*)	-	-
Assertion Uses	1* (1*)	2* (2*)	-	-
`run` Commands	1 (1)	1 (1)	47.6% (63.6%) } = 100%	$[0, 4]$ $([0, 5])$
`check` Commands	1* (1*)	0 (0)	52.4% (36.4%)	$[0, 4]$ $([0, 2])$
Set Cardinality Operator (#)	2* (1)	6* (4*)	-	$[2, 27]$ $([1, 20])$
Transitive Closure Operators (^ and $\star$)	1* (2*)	6* (9*)	-	$[0, 9]$ $([0, 25])$
Partial Functions	1* (1*)	0 (0)	12.3% (16.0%) } of all fields	$[0, 2]$ $([0, 2])$
Total Functions	1* (3*)	2 (2)	53.6% (34.0%)	$[0, 10]$ $([0, 5])$

Approach & Findings: The results of this research questions are shown in Table 1 and we describe them below.

MODEL LENGTH: The simplest characteristic of any model is its length. 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. For multi-file models, we count the number of lines across all user-created files that make up each multi-file model. We find that the predominant value (*i.e.,* mode) for model length in the general corpus is 52 lines, whereas the typical value (*i.e.,* median) is 63. The expert predominant and typical use values are slightly higher coming in at 73 and 76 lines respectively. Similarly, the common range for expert model length is also higher than its general corpus counterpart. *We conclude that expert modelers tend to write longer Alloy models. Overall, Alloy models are fairly short especially when compared to programs.*

MODEL SPAN: We also compute the span of Alloy models by counting the number of user-created files that make up each model. We report the predominant and typical use criteria, which represent the most frequent number of files in a model and the central tendency of the model span respectively. We find that the predominant and typical values for model span are both one. *Hence, Alloy models are typically made up of one file only.*

SIGNATURES AND FIELDS: We present a number of data summary criteria related to signatures in Alloy models including the signature count per model and the distribution among top-level, subset and extension signatures. Our results indicate that a typical Alloy model contains eight signatures, although the predominant use value is two. *The signature count in Alloy models exhibits a great deal of variation as shown by common range ($[2, 25]$).* Examining expert models yields similar results, which shows that expert models conform to this trend of variation in signature counts.

Top-level signatures create mutually disjoint sets that are not subsets of another set. **Subset signatures** are declared as subsets of another signature using the keyword `in`. 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`. Signature extensions can also be introduced using `enum` .

We find that top-level signatures account for 22.6% of all signatures in the general corpus. Subsignature extensions are the most prominent kind of signatures coming in at 73.9%. Subset signatures are quite sparse in Alloy models

(1.5% of all signatures). The typical Alloy model contains four top-level signatures and three subsignature extensions but no subset signatures. The disparity between the typical use values and percentage distribution for top-level and subsignature extensions is due to the skewness of the produced data set of signature counts. The percentage distribution of signatures by level in the expert corpus shows a relatively similar trend. Top-level signatures account for 32.5% of all signatures in the expert corpus, whereas subset signatures and subsignature extensions constitute 0.3% and 60.5% of all signatures respectively. *Therefore, subsignature extensions (partitions of the universe) are used abundantly and subset signatures are not often used.* We conclude that the most common use of set hierarchy is to partition the universe. **Action item:** Tool developers may want to create optimizations centered around the common use of set hierarchy to partition the universe.

Alloy has no construct for scalars. Instead, modelers declare signatures that have multiplicity `one` (a set of size one) or alternatively, they use the `enum` construct, which is syntactic sugar for declaring each listed element as a subsignature extension with multiplicity `one`. While the keyword `one` can be used to instantiate any kind of signature with multiplicity `one` (*i.e.,* top-level, subset or subsignature extension), `enum`s can only be used create ordered subsignature extensions with multiplicity `one`. 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. We count signatures declared with the keyword `one` and `enum`s. The typical Alloy model contains one signature declared with multiplicity `one`. Signatures with multiplicity `one` account for 26.4% of all signatures declared across all the models in the general corpus. Signatures with multiplicity `one` account for a significantly higher share of all signatures in the expert corpus coming in at 41.5%. *The significant use of scalars in Alloy models is evident.* **Action item:** Tool developers should ensure that they utilize any optimizations for sets that are scalars (such as converting them to scalars in an SMT solver).

We find that `enum`s account for 1.9% of all signatures in the general corpus. The typical use criterion for `enum`s is zero, which means that the typical Alloy model does not contain an `enum` declaration. We find that `enum`s account for 6.7% of all signatures in the expert corpus, which shows that *expert modelers tend to use `enum`s significantly more often.* Nevertheless, scalars declared using signatures with multiplicity `one` far outnumber the ones declared using `enum`s in both corpora. *`enum`s are an underutilized construct in Alloy.* **Action item:** Educators are encouraged to highlight the use of `enum`s to concisely instantiate multiple ordered subsignature extensions with multiplicity `one`.

An **`abstract` signature** has no elements except those belonging to its extensions or subsets. *We find that `abstract` signatures are fairly uncommon in Alloy model.* The typical Alloy model contains only one `abstract` signature. We also find that `abstract` signatures account for 10.3% of the total number of signatures across all models in the general corpus. The use of `abstract` signatures in the expert corpus does not differ significantly from their use in the general corpus.

We find that in a typical Alloy model, signatures with field declarations typically have two fields. We also find that that signatures commonly have between one and four fields which shows that *modelers are not aggregating fields under one signature but spreading them over multiple signatures.* The vast majority of signatures in the general corpus do not have fields (82.2%) while only 17.8% of signatures have fields associated with them. Expert models tend to have significantly more signatures with fields (29.1% of all signatures). We also examine the use of fields among the different signature kinds. We find that 57.8% of top-level signature have fields (compared to 56.4% in the expert corpus). We find that 27.7% of subset signatures have fields in the general corpus, but none of the subset signatures in the expert corpus have fields. Only 7.2% of signatures with multiplicity `one` (compared to 4.1% in the expert corpus) have fields (whether they are top-level of not). Similarly, we find that only 5.9% of subsignature extensions have fields in the general corpus. This percentage increases significantly for the expert corpus where 16.6% of subsignature extensions have fields. Given that there are relatively few subset signatures, it is clear that *most fields are declared with top-level (non-multiplicity-one) signatures.*

FORMULAS: Next, we quantify formula use in Alloy models by counting **top-level formulas**, *i.e.,* formulas that are not part of any other larger formula. Our results indicate that a typical Alloy model contains 20 top-level formulas. *Akin to the signature count, formula use in Alloy differs significantly from one model to another as shown by the common range* $[3, 93]$. Our results indicate that expert models have considerably higher formula counts with the predominant and typical use values coming in at 10 and 34 respectively. *The common range for the formula count in the expert corpus (*$[6, 227]$*) shows the same variation observed in the general corpus albeit with considerably higher lower and upper bounds.*

FACTS: We assess the use of fact blocks and (non-empty) signature fact blocks. Jackson [11] states that the use of signature facts should be limited since the implicit quantification can lead to unexpected consequences. We find that in the general corpus, 47.6% of signatures have a fact block associated with them and 41.0% of all fact blocks are signature fact blocks. The common range of $[0, 3]$ indicates that in 75% of models the number of signatures with fact blocks is between zero and three. *Signature facts are used extensively in Alloy models in the general corpus.* Only 9.5% of signatures in the expert corpus have a fact block associated with them and signature facts in these models account for 15.3% of fact blocks only. These results are in sharp contrast to the ones obtained when examining the general corpus and show that *expert modelers tend to avoid using signature fact blocks.* We hypothesize that expert modelers are more familiar with the implicit quantification associated with signature facts and thus use them sparsely to avoid erroneous results. **Action items:** 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.

FORMULA CONTAINERS: We present a number of data summary criteria pertaining to the use of formula containers in Alloy models including predicate, function and assertion declarations and calls. When a function is used with a `run` command, Alloy finds an instance that makes the formula within the command true [39]. In this case, the instance consists of a collection of arguments for the function, the values of signatures and fields, and the function result. The distribution of predicate and function calls across commands and formulas is particularly interesting because it gives us an insight into how modelers are using these formula containers. Predicates and functions used with command queries are utilized for model verification whereas the ones called in formulas are used for model description.

We find that 86.3% of all models in our general corpus contain predicate declarations compared to 35.4% of models that contain function declarations. The predominant and typical use values show a greater use of predicates than functions among Alloy modelers in the general corpus. *We find that modelers declare and call predicates significantly more often than functions.* A typical Alloy model contains four predicate declarations and four predicate calls but no function declarations or calls. This trend of predicate prevalence is also reflected in the percentage distribution where 77.4% of these parametrized declarations are predicates while the remaining 22.6% are functions. Similarly, predicate calls account for 75.0% of the total parametrized expression calls, with function calls accounting for the remaining 25.0%. We find that predicate use in formulas (91.6%) greatly outnumbers predicate use in commands (8.4%). Functions are almost exclusively used in formulas. Predicate prevalence is still observed in the expert corpus albeit to a considerably lesser extent. 89.3% of expert models have predicate declarations whereas 60% of expert models contain function declarations which is a significant increase over the general corpus. *Thus, expert modelers are more likely to create and use functions in their models.* We find that 63.1% of parametrized declarations in expert models are predicates while the remaining 36.9% are functions. Similarly, predicate calls account for 57.0% of the total parametrized expression calls, with function calls accounting for the remaining 43.0%. *We conclude that 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 in both corpora.* We also find that *using functions with a `run` command is an underutilized functionality of the Alloy language.* **Action item:** Educators are encouraged to highlight this functionality and explain to student modelers how it can be used to obtain a collection of values for the arguments of a function, the values of signatures and fields, and the function result.

We find that 40.7% of Alloy models in the general corpus contain assertion declarations compared to 37.3% in the expert one. 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. Assertions are remnants of an older version of Alloy and can be essentially thought of as unparametrized predicates [40]. *We conclude that assertions are used in Alloy models even though they are a remnant of an older version of the language.* **Action item:** Language designers may wish to discuss whether the keyword `assert` is useful to distinguish a particular kind of predicate or whether it has become a redundant construct and should be removed.

Commands: We investigate the number of command queries (`run` and `check`) in a model. 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. The typical use criterion per model for `check` commands in the general corpus is zero whereas its `run` counterpart is one. A typical Alloy model contains one `run` command but no `check` commands. The percentage distribution is near-equal between `run` and `check` with 47.6% of queries being `run` commands while the remaining 52.4% of queries are `check` commands in the general corpus. *Therefore, `run` and `check` commands are equally valuable to modelers.* The expert corpus results deviate significantly from the general corpus results and show *a clear preference for `run` commands over `check` commands among expert modelers* given that 63.6% of commands in the expert corpus are `run` commands wheres the remaining 36.4% are `check` commands.

Advanced operators: We explore the use of set cardinality and transitive closure in Alloy models. The use of these advanced operators involves extensive expansion of formulas and thus can affect analysis complexity and solving time. The **set cardinality operator** (#) allows modelers to specify the size of a set consisting of a signature, field or set expression. We extract from the model all uses of the set cardinality operator. We scale the number of set cardinality operators in a predicate or function according to the number of calls corresponding to that predicate or function. We find that 44.2% of all models in our general corpus (and 42.6% of models in the expert corpus) have at least one use of the set cardinality operator. Alloy models that uses set cardinality in the general corpus contain six uses of this operator. We find that expert models have a slightly lower use frequency for the set cardinality operator. *Overall, the set cardinality operator count can vary significantly among models that make use of it since the non-zero common range in the general corpus is $[2, 27]$ and $[1, 20]$ in the expert corpus.* **Action item:** Tool developers are encouraged to create solvers and optimizations that address the abundant use of the set cardinality operator in Alloy.

We repeat this process for the **transitive closure operators** in Alloy (^ and $*$). We find that 35.8% of models in the general corpus (and 33.3% of expert models) contain at least one transitive closure operator. Models that make use of transitive closure operators in the general corpus typically contain six uses of these operators (scaled by function and predicate calls). Expert models that make use of transitive closure have a slightly higher use frequency for these advanced operators. *Thus, transitive closure operators are used abundantly in models that require them.* **Action item:** We encourage tool developers to explore developing optimizations centered around the transitive closure operators.

Partial and Total Functions: We explore the use of total and partial functions in Alloy models. A function (compared to a relation) can be handled differently by different solvers. For example, KodKod [33] represents total functions as relations with additional constraints, whereas Portus [30] (an SMT-based finite model finder for Alloy) represents total functions as total functions for SMT solving. Partial functions in Alloy are declared as fields under signatures (*e.g.*, `f: e1 -> lone e2` and `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 field that maps every element in the domain to some element in the range (*e.g.*, `f: e`, `f: one e` and `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.

We find that partial functions account for 12.3% of all fields whereas total functions account for 53.6% of all fields. However, user-introduced total functions over subsets are really partial functions. Hence, optimizations that target total functions may be limited to user-introduced total functions over top-level signatures. We find that 58.60% of all user-introduced total functions (*i.e.*, 31.1% of all fields) are over top-level signatures. *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.* It is likely that these percentages would be higher if we included functions in library modules. **Action item:** It is worthwhile for tool developers to work on improving the analysis methods with a focus on total functions.

RQ# 2: 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 quite a lot of variation in the length of Alloy models as shown by the common range. 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 the general 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 number of top-level formulas in the body of predicates and functions is scaled according to the number of calls made to the formula holder (predicate, *etc.*). We filter out models that have a set or top-level formula count of zero to ensure that the logarithmic transformation can be applied. This procedure limits our general corpus size to 1,561 models out of 1,652 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 (Residuals vs. Fitted, Normal Q-Q, Scale-Location and Residuals vs. Leverage) for each linear regression model (which can be found in Eid [41]). We repeat this process for the expert corpus.

Findings: Fig. 1 shows best fit line for the Model Length vs. the Number of Sets in the models of the general corpus. The r value of 0.77 suggests a high positive correlation between the set count and the model length. However, with a conservative goodness of fit ($R^2 = 0.59$), this correlation only explains 59% of the variation in the data. The residual plots conform to the established characteristics of a good linear regression model fit which indicates that the fit cannot be significantly improved beyond $R^2 = 0.59$. Since the coefficient of determination R^2 is relatively conservative,

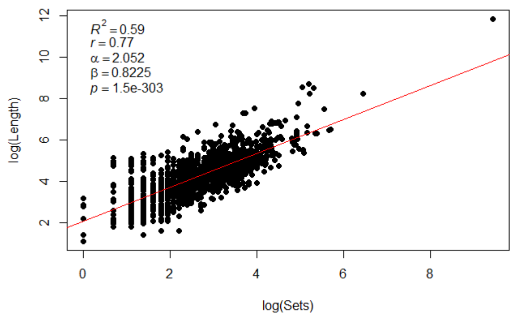

Fig. 1. Model Length vs. Number of Sets in log scale

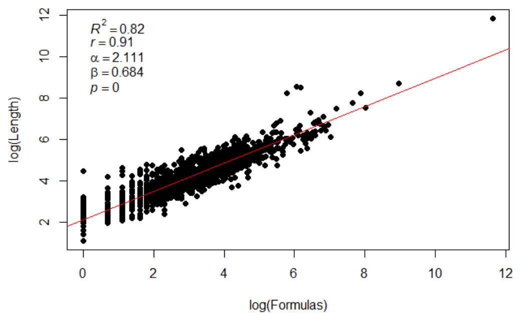

Fig. 2. Model Length vs. Number of Formulas in log scale

we conclude that the high correlation between set count and model length is not applicable to a large portion (41%) of the subset of models used in this research question. The best fit line obtained from the expert corpus shows an even lower correlation between model length and the number of sets ($r = 0.67$) and is coupled with lower goodness of fit $R^2 = 0.45$. *Therefore, set count is not a good predictor for model length.*

We attempted to correlate model length with the number of top-level signatures in the model, but the produced linear regression models indicated a negligible correlation between these two characteristics.

Fig. 2 shows the best fit line for the Model Length vs. the Number of formulas in the general corpus models. We find that $r = 0.91$ which indicates a strong positive correlation between formula count and model length that is applicable to 82% of data points ($R^2 = 0.82$). All the residual plots indicate that the fit cannot be significantly improved. The best fit line for the expert corpus produces similar results ($r = 0.88$ and $R^2 = 0.77$). *Hence, formula count is a good predictor for model length (and better than set count). Longer Alloy models will probably have more formulas but not necessarily more sets.*

6 Modeling Practices

This section investigates research questions that have to do with how modelers use the language's constructs and how they express descriptions in Alloy. These research questions are far more intricate and require multiple complex queries and several traversals through the parse tree. These "deeper-level" questions often require considerable post-processing and utilize several external data structures.

RQ# 3: How commonplace is the use of modules in Alloy?
Motivation: 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/fields/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 and classify modules as user-created (*i.e.,* they do not contain the string `util`) and standard library modules (*i.e.,* they contain the string `util`). If a library is parametrized, each instance of the use of a library is counted[3]. The use of integers in Alloy (`Int` set or numeric constants) does not require opening the integer library unless arithmetic operators are used. 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. We present the distribution over `open` (if applicable) and over models. A multi-file model imports a `util` module if at least one file in the model imports that module.

TABLE 2
Usage of User-Created and Library Modules

Library module	Distribution over `open` Statements	Distribution over Models
user-created	31.4% (34.8%)	15.1% (17.3%)
ordering	41.2% (45.2%)	31.0% (52.0%)
integer	14.2% (0.0%)	14.6% (0.0%)
integer w/o import	-	52.1% (38.7%)
boolean	8.1% (4.5%)	7.6% (9.3%)
relation	2.8% (7.1%)	2.7% (10.7%)
ternary	0.7% (0.0%)	0.7% (0.0%)
graph	0.7% (3.9%)	0.6% (6.7%)
naturals	0.5% (4.5%)	0.5% (9.3%)
seq	0.2% (0.0%)	0.2% (0.0%)
time	0.2% (0.0%)	0.2% (0.0%)
seqrel	0.0% (0.0%)	0.0% (0.0%)
sequence	0.0% (0.0%)	0.0% (0.0%)

Findings: Table 2 shows the distribution of user-created and library modules over `open` statements and over models. User-created modules account for 31.4% of all `open` statements in the general corpus. We also find that 15.1% of all models in the general corpus contain `open` statements that import user-created modules *i.e.,* 15.1% of models in the general corpus are multi-file models. 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 41.2% of `open` statements and 31.0% of models in the general corpus. The integer and boolean modules come in second and third place respectively. We found no uses of the `seqrel` and `sequence` modules. Integers are used abundantly in ways that do not involve importing the integer module. In total, 66.7% of all models use integers but only 14.2% of models import the integer library module explicitly. We find that 17.3% of expert models include an `open` statement that imports a user-created module *i.e.,* 17.3% of expert models are multi-file models. The ordering module accounts for 45.2% of `open` statements in expert models and is used in 52.0% of models in the expert corpus. We did not find any expert model that explicitly imports integers even though 38.7% of expert models use integers. *The expert results show a greater use frequency for most library modules. However, expert modelers are less likely to import the integer module and use integers in their models. Overall, we find that `open` statements for user-created modules are relatively prevalent in Alloy models and that some library modules such as the ordering module and integers are used extensively whereas others are rarely used.* **Action items:**

- Environment designers should consider adding multi-file model management features to their IDEs.
- Educators can better highlight the value of underutilized library modules.
- Optimizations (such as symmetry breaking constraints) exist for the ordering module, however, given its prevalent use, tool developers are encouraged to continue to investigate optimizations for the ordering module and for integers in Alloy.

RQ# 4: How are integers used in Alloy models?
Motivation: Integer use is often discouraged in Alloy. According to Jackson [11], most problems with integer values do not require integers to be modeled and would benefit from more abstract descriptions and constraints. 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.*). Alloy provides rudimentary support for arithmetic operations through the integer module. Modelers also 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. We examine the use of integers in Alloy to help educators determine how much to emphasize alternative modeling techniques that can replace integers.

Approach: We tally the number of different integer constants and the number of times the integer set (`Int`) is used as an argument in a field declaration within each model to determine the typical use and predominant use per model. For the distribution, we partition the sums of these values over all models into the field arguments that are declared as integers and the number of different integer constants. For example, the constant '`1`' is counted once for each file that it is used within. We then examine the uses of each constant and each field (within one file) to determine if it is used exclusively with relational operators (meaning no arithmetic operations and no set cardinality). If it is, then the constant/field could be replaced in that model with an element from an ordered set rather than using the set of integers. This substitution is not possible for integer fields/constants used with arithmetic operators. We also determine

3. Implicit imports of the ordering module via the use of `enum` declarations are also counted.

if the constant/field is used only with relational operators and set cardinality. We revisit what can be done with the integer fields/constants that are used with set cardinality (and not arithmetic) in the next research question.

TABLE 3
Integers in a Model (Non-Zero Values)
(PU = Predominant Use, TU = Typical Use)

Use	PU	TU	Percentage Distribution	
Fields	1 (1)	2 (1)	29.8% (10.3%)	
w/ only `</<=/=</=/>/>=/=>`	1 (1)	1 (1)	54.9 (60)	
w/ set cardinality	1 (0)	1 (0)	1.4 (0)	= 100%
w/ arith ops	1 (1)	1 (1)	22.1 (10)	
other			21.6 (30)	
Constants	1 (1)	2 (2)	70.2% (89.7%)	
w/ only `</<=/=</=/>/>=/=>`	1 (1)	2 (2)	26.6 (22.2)	
w/ set cardinality	1 (1)	2 (1)	50.7 (58.0)	= 100%
w/ arith ops or other uses	1 (1)	2 (1)	22.7 (19.8)	

Findings: Table 3 shows the data summary criteria for integers in Alloy models. We find that constants account for the overwhelming majority of integers (70.2%) in the general corpus, whereas integers in field declarations constitute only 29.8% of integers across all models. This percentage distribution differs significantly in the expert corpus with fields accounting for 10.3% of integer use only with the remaining 89.7% of integers being constants. We find that 54.9% of all integers used as arguments in fields in the general corpus do not need to be integers (meaning they are used exclusively with relational operators) and could be a set with a linear ordering. This optimization is not possible for 23.5% of all integer fields since they are used with the set cardinality operator or with arithmetic operators. The remaining 21.6% of integer fields were not used with relational or arithmetic operators in the model post declaration. We did not find any instances where expert modelers were using integer fields with the set cardinality operator. The next research question examines the use of set cardinality in more detail. With respect to the use of the constant integers in the general corpus, 26.6% are use exclusively with relational operators, 50.7% are used with the relational operators and set cardinality, and the remaining 22.7% account for integer constants used in macros, with arithmetic operators or passed as arguments. The ones used exclusively with relational operators can be replaced with constants of a set with a linear ordering. The percentage distribution of integer constants use in the expert corpus is similar to the one produced by examining the general corpus. *We conclude that Alloy modelers are using integers in many places where they could be using the ordering module.* **Action items:** This finding means that there are opportunities to improve Alloy models to make their analysis more efficient, for example:

- Language designers could add a built-in set and identifiers that model numeric constants that are a linear order (*e.g.,* `One`, `Two`, *etc.*) to provide the intuition of using integers as a linear ordering without actually using integers for analysis.
- Educators can promote the use of the ordering module.
- Environment designers can modify the Alloy Analyzer to allow it to warn users about integer constants/fields that are only used as a linear order.
- Tool developers can create optimizations to convert many integer uses to an application of the ordering module before analysis.

RQ# 5: Is the set cardinality operator used with integer constants 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 either command query scopes or multiplicity keywords in the signature declarations. Setting set sizes using the cardinality operator often results in slower solving times [42]. Jackson also discourages the use of the set cardinality operator with integers to designate the size of a signature set [11].

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: 1) expressions that can be turned into scope limitations and 2) 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>`) and compared with a constant number.

The set cardinality operator applied to a signature set with the operators </>/>= or the set cardinality operator applied a set expression with any relational operator (*i.e.,* `# <sig> </>/>= <num>` or `# <set expression> </>/>=/ =/<=/=> <num>`) and compared with a constant number can be turned into formulas. For instance, if `A_member` is a predicate used to denote set membership in set `A`, then the expression `# A = 2` can be replaced by an internal optimization with 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 [30].

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.

Findings: The results of this research question are presented in Tables 4 and 5. We find that 72.1% of all set cardinality operators in the general corpus are either used with numeric operators or with a combination of numeric and relational operators and thus cannot be converted to scope limitations

TABLE 4
Uses of Set Cardinality

Expression	Percentage Distribution
Set Cardinality without Relational Operators	72.1% (95.8%)
Set Cardinality with Relational Operators (broken down below)	27.9% (4.2%)
`# <sig> =/=</<= <num>`	12.9% (2.9%)
`# <sig> </>/>= <num>`	3.8% (0.3%)
`# <set expr> </>/>=/=/<=/=> <num>`	11.2% (1.0%)

TABLE 5
Percentile Distribution of Integer Constants in Set Cardinality Uses

Construct	12.5th	25th	50th	75th	87.5th
Integers - General	1	1	2	2	4
Integers - Expert	1	1	3	3	5

Common Range - General : $[1, 4]$
Common Range - Expert : $[1, 5]$

or formulas. Set cardinality uses that specify the size of a set using constant integers account for 27.9% 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 15% of all set cardinality applications in the general corpus. Table 5 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 and converting them to formulas is reasonable. *Our results show that the use of set cardinality to specify the size of a set instead of using multiplicity keywords, command queries, or formulas is considerable, but not abundant.* The use of set cardinality in the expert corpus differs significantly compared to the entire corpus. *We find that expert modelers are significantly less likely to specify the size of a set using the set cardinality operator.* The vast majority of set cardinality operators in the expert corpus are used with numeric operators or with a combination of numeric and relational operator with only 4.2% of set cardinality operators being used to specify the size of a set. Thus, only 3.2% of set cardinality applications in expert models are expressions that can be turned into formulas. **Action items:** Environment designers can include a warning that discourages the use of set cardinality to set the size of sets. Educators can highlight the proper use of multiplicity keywords and setting scopes in command queries. Tool developers can create optimizations to internally transform these uses of set cardinality into signature declarations with a multiplicity keyword, into command queries, or into formulas.

RQ# 6: 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 [11]. In the **predicate calculus style** (shown on line 1 of Fig. 3), quantifiers over variables are used along with Boolean operators (but no set operators). The same formula can be written in the **relational calculus style** (shown on line 2) where expressions denote relations, and multiplicity operators on relations are used to accomplish quantification. The third style is called the **navigation expression style** (shown on line 3) where 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 has the advantage of conciseness. The three styles do not have equivalent expressive power. The navigational style is the most expressive amongst them because the predicate and relational calculus styles lack transitive closure and quantifiers respectively. By answering this question, we help educators determine how much to emphasize the distinction between the different styles in teaching. 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
```

Fig. 3. Three Formula Modeling Styles

TABLE 6
Model Classification by Formula Style

Model Classification	Distribution
Pure Relational Calculus	18.6% (24.0%)
Dominant Relational Calculus	52.9% (68.0%)
Pure Navigation Expression	6.1% (0.0%)
Dominant Navigation Expression	21.7% (8.0%)
Pure Predicate Calculus	0.3% (0.0%)
Dominant Predicate Calculus	0.4% (0.0%)

Approach: We extract formulas contained within the body of facts, predicates, functions, assertions and macros. 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. 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 in a model (including multiple files where applicable). After processing all the formulas in a model[4], 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,

4. In this case, we do not scale for use in predicates/functions.

the model gets labeled with the dominant writing style, *i.e.,* the style matching the largest number of formulas in that model.

Findings: Table 6 shows that in the general corpus, 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. Relational calculus is by far the most prominent formula style in the expert corpus given that 68.0% of expert models fall in the dominant relational calculus category and 24.0% of expert models pertain to the pure relational calculus category. The remaining 8.0% of expert models fall in the dominant navigation expression category. We did not find any expert models that belong in the predicate calculus categories. *We conclude that expert modelers have a clear preference for 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.* **Action item:** Educators should consider the frequency of use of the different formula modeling styles when deciding which modeling style to teach first or emphasize.

RQ# 7: How deep/wide are the set hierarchy graphs in Alloy models?

Motivation: **Subsignature extensions** allow modelers to introduce 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 where each parent signature is denoted as a node with one or more child nodes that represent its subsignature extensions. **Subset signatures**, using the keyword `in`, 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. By building and exploring the characteristics of extension and subset hierarchy graphs (depth and width), we can get a better understanding of how modelers create sets and extend them.

Approach: We build the **extension hierarchy graph** iteratively over multiple steps. If a model consists of multiple files, we include all signature declarations across all files that make up a single model when building the hierarchy graphs. We build the hierarchy graph by adding nodes corresponding to each parent signature and its extensions (including those declared using `enum`) and creating edges between them. We also add to the graph any remaining top-level signatures that correspond to signatures that are not extended after being declared. Once the hierarchy graph is built, we compute its depth and width. The depth of an extension hierarchy graph is the number of edges on the longest downward path between a top-level parent node and a leaf node. The width of an extension hierarchy graph is the number of extension leaf nodes (*i.e.,* excluding leaf nodes that correspond to top-level signatures).

The **subset hierarchy graph** is an augmentation of the extension hierarchy graph obtained by adding subset nodes to the existing graph. 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. We do not compute the width of the subset hierarchy graph because subsets can overlap.

TABLE 7
Depth and Width of Extension Hierarchy Graphs
(An asterisk * indicates a non-zero criterion)

Measure	**Predominant Use** (Mode)	**Typical Use** (Median)
Depth	1 (1)	1 (1)
Width	2* (4*)	3 (4)

TABLE 8
Percentile Distribution of Extension Hierarchy Graph Depth and Width

Measure	**12.5th**	**25th**	**50th**	**75th**	**87.5th**	**CR**
Depth - General	0	0	1	1	2	$[0, 2]$
Depth - Expert	0	0	1	2	2	$[0, 2]$
Width - General	0	0	3	8	15	$[0, 15]$
Width - Expert	0	0	4	8	21	$[0, 21]$

Findings: Tables 7 and 8 show the predominant and typical use values and percentile distribution of the extension graph depth and width respectively in both corpora. We find that the predominant and typical value for the depth of the extension hierarchy graph are one in both corpora. Hence, in a typical Alloy model, signatures are not extended beyond a single level. The common range of extension hierarchy graph depth is $[0, 2]$ which means that 75% of depth values are either zero, one or two and more specifically 62.5% of depth values are zero or one. *Overall, extension hierarchy graphs in Alloy models are very shallow.* The non-zero predominant use value for the width of the extension hierarchy graphs is two whereas the typical use value is three. The percentile distribution of the width exhibits a significant amount of variability with the common range being $[0, 15]$. We conclude that 75% of all width values fall between zero and fifteen. The expert common range for width is relatively similar to the general one albeit with a higher upper bound of twenty-one. *The extension hierarchy graphs in the expert corpus are comparable in depth to the ones generated from the general corpus but they are slightly wider.*

Tables 9 and 10 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 in the general and expert corpora, 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. *Thus,*

TABLE 9
Depth of Subset Hierarchy Graphs
(An asterisk * indicates a non-zero criterion)

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

TABLE 10
Percentile Distribution of Non-Zero Subset Hierarchy Graph Depth

Measure	**12.5th**	**25th**	**50th**	**75th**	**87.5th**	**CR**
Depth - General	1	1	1	1	1	[1, 1]
Depth - Expert	1	1	1	1	1	[1, 1]

creating subsets of a subset are rare occurrences in Alloy models.
Action items:

- Tool developers do not need to ensure that their optimizations scale favorably for deep extension hierarchies.
- Optimizations for the language do not necessarily need to account for subsets of subsets.
- Optimizations for encoding sets for solving may be able to take advantage of the fact that the set hierarchies in Alloy models are mostly shallow.

RQ# 8: 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 [43] using metrics such as the Lack of Cohesion of Methods (LCOM) metric [25]. 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 to measure the cohesion of an Alloy model. We define the cohesion of an Alloy model as the degree of connectedness between its components.

Approach: The **Signature Connectedness Graph** or **SCG** is a measure of the number of strongly connected components of a model's signature graph. This graph shows 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. If a model consists of multiple files, the SCG of that model includes all signatures declared in all the files that make up that model as well as any connections that exist between them.

```
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}
```

Fig. 4. Example of Alloy Signature Declarations

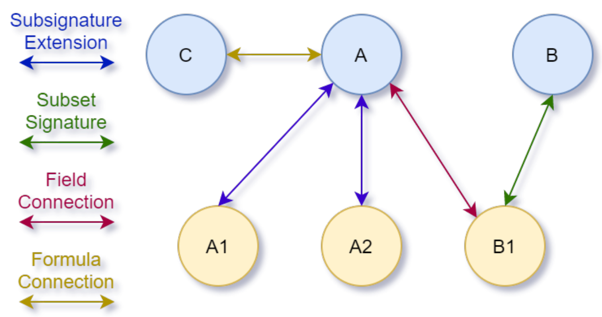

Fig. 5. SCG for Fig. 4 with SCG = 1

Fig. 4 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 a top-level signature declared on line 5. `A1` and `A2` are subsignature extensions of `A` declared on line 4. The signature fact block associated with `C` contains a reference to the field `f1` of `A`. Fig. 5 contains the SCG corresponding to the signature declarations in Fig. 4. 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 Fig. 5, 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.

If we remove the formula `no iden & f1` from the signature fact body of `C` then the `C` node in the graph will no longer be connected to `A`. The resulting graph would contain 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 that graph would be two.

Findings: Tables 11 and 12 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. Examining the SCG metric results for expert models yields identical results. *Our results indicate that Alloy models are highly cohesive given that the overwhelming majority of models in our corpus had an SCG metric value of one.* We believe that modelers may benefit from having access to the SCG to understand their models. **Action items:** Environment designers may want to consider creating a tool that generates the SCG graph and metric for Alloy models and warn users if their SCG is not equal to one.

TABLE 11
SCG Metric Value

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

TABLE 12
Percentile Distribution of SCG Metric Values

Computation	**12.5th**	**25th**	**50th**	**75th**	**87.5th**
SCG - General	1	1	1	1	2
SCG - Expert	1	1	1	1	2

Common Range - General : $[1, 2]$
Common Range - Expert : $[1, 2]$

RQ# 9: 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. Also, 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+.

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 then we identify whether each signature name is only used with box or dot join expressions involving one of its fields. We ignore the use of a signature by itself as a parameter type in a predicate or function (because these signatures are used within formulas). Expressions that contain an application of the transitive closure operators (*e.g.,* `A.^f1` or `A.*f1`) are excluded from this count. For multi-file models, we examine the use of each signature in all files that make up a single model to determine if that signature is being used as a structure. We report the total number of signatures used as structures for each model.

Findings: We find that 10.3% of all signatures with fields in the general corpus are used only as structures. We also find that 28.1% of models in the general corpus contain at least one signature that is used as a structure. In expert models, 6.9% of signatures with fields are used as structures and 18.6% of expert models contain at least one signature used as a structure. *While not abundant, signatures used as structures are still present in a considerable portion of Alloy models.* **Action item:** Language designers may want to consider adding support for structures in the Alloy language.

RQ# 10: 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). We also explore the use of the ordering module with a non-exact scope. The ordering module forces the signature on which it was instantiated to be exact [11]. 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. An illustrative example of the problematic implications of using ordered sets with non-exact scopes can be found on Stack Overflow [44]. Lastly, we examine how integer scopes are set 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: 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 subsets with no scopes associated with them, the scopes of the subsets are derived to be equal to that of the parent signature. Similarly, if a top-level signature has a set scope and all its signature extensions (mutually disjoint) except for one have scopes associated with them, then the missing scope of the subset can be derived. On the other hand, if the top-level signature does not have a set scope, but its child signature extensions have scopes associated with them, then the parent signature has a derived scope equal to the sum of the scopes of its child signatures. 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.

Next, we identify ordered signature sets that are used with non-exact scopes. 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. We also extract and tally up all command queries containing an integer scope specification.

TABLE 13
Scope Categories Across All Commands and All Models

Scope Category	Distribution
Default Non-Exact	55.2% (2.1%)
Set Non-Exact	15.8% (51.7%)
Model Exact	11.6% (12.8%)
Set Exact	10.6% (10.8%)
Derived Non-Exact	4.9% (18.5%)
Derived Exact	1.9% (4.1%)

Findings: Table 13 shows that a little over half of all `run` and `check` commands in the general corpus fall in the default non-exact category. Set non-exact is the second most populous category in the general corpus coming in at 15.8%. 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. *Overall, 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.* The distribution of commands across the scope categories differs significantly in expert models compared to the general corpus. We find that 51.7% of commands in the expert corpus fall in the set non-exact category. The derived non-exact category is the second most populous category for commands in expert models. *Thus, expert models are more likely to specify set scopes in `run` and `check` commands. Expert modelers are also significantly more likely to take advantage of Alloy's scope derivation.* **Action item:** Educators should discuss when/how to set scopes and explain when exact scopes are appropriate.

We find that modelers are frequently applying the ordering module to a set with a non-exact scope given that 52.4% of all ordered sets in the general corpus have a non-exact scope (compared to 55.5% in the expert corpus). Table 14 shows the distribution of ordered sets across the non-exact scope categories. The vast majority of ordered sets with a non-exact scope in the general corpus fall under the set

TABLE 14
Scope Category Distribution Among Ordered Sets

Scope Category	Percentage Distribution	
Set Non-Exact	81.4% (55.5%)	of the 52.4% (55.5%)
Default Non-Exact	17.9% (0.0%)	
Derived Non-Exact	0.7% (0.0%)	

non-exact scope category (81.4%) followed by the default non-exact scope category (17.9%). Only 0.7% of ordered sets with a non-exact scope fall under the derived non-exact category. All the ordered sets with a non-exact scope in the expert corpus fall under the default non-exact category. *Thus, modelers are explicitly setting non-exact scopes for ordered sets*. Given that commenting out command queries is a fairly common practice among Alloy modelers, we hypothesize that these results are an underestimate of the actual frequency of occurrence of this practice. **Action items:** Educators are encouraged to highlight this bad modeling practice and explain its repercussions on the command results. Environment 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.

We find that integer scope specifications account for 7.2% of the total number of scoped sets in commands in the general corpus. In expert models, integer scope specification constitute only 2.9% of scoped sets in commands. A typical Alloy model in the general corpus sets the scope of integers to 5, whereas a typical expert model sets the integer scope to 6. The most common integer scope value is 5 in the general corpus and 6 the expert corpus. We find that the common range for integer scopes in the general corpus is $[3, 8]$, *i.e.,* 75% of integer scope values fall in that range. *Expert models tend to use even higher integer scopes* given that the expert common range for integer scopes is $[5, 8]$. *We find that modelers often specify higher integer scopes than the default scope.* 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. **Action items:** Environment designers may want to reconsider the default scope for integers in Alloy. Educators should familiarize students with proper integer scope use and help them find alternative ways of modeling problems with numeric constants without using integers.

7 Threats to Validity

Our results in evaluating a set of research questions on a corpus of Alloy models allow 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 acknowledge that the subset of expert models used in this study was selected by hand and thus the conclusions drawn from examining expert models may not be applicable to all expert modelers. We note that we did 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.*). We also acknowledge that some models in our corpus may be automatically generated for testing analysis techniques and thus the scopes used in some commands may be artificially inflated to test if these analysis techniques scale to larger scopes.

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 parser was tested extensively to ensure that it can properly parse any syntactically correct model written in 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. We also acknowledge that our results do not include counting the constructs within a library module, which may alter our results.

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 Related Work

Wang *et al.* [45] 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 propositional logic (SAT) model. The tally of these features on 119 Alloy models 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 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 [46] 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 aims to understand how people write Alloy models (rather than solver performance). We look at a more general set of Alloy models (1,652 models) and draw conclusions from larger patterns within the model's syntax. We provide actionable items based on our findings.

Zheng *et al.* [47] created a corpus of auto-generated Alloy models used to evaluate the performance of Platinum, an extension of the Alloy Analyzer that supports efficient analysis of evolving Alloy specifications. These models were created by applying a series of mutation operators to existing Alloy models in order to generate multiple modified versions of these specifications used for performance analysis. Because these auto-generated models do not represent a typical Alloy model and disproportionally represent one modelling style, we did not include them in our corpus.

Chowdhury and Zulkernine [25] use complexity, cohesion and coupling metrics as a means to predicate software vulnerability in programs. We have drawn inspiration from Chowdhury and Zulkernine's work when formulating some of our research questions 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. We have also drawn inspiration from object-oriented programming and UML modeling metrics such as those described in Briand and Jüst [23] and SDMetrics [24].

Lopes and Ossher [48] 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 and scale and these measures have significant implications on object-oriented software metrics that do not currently account for differences in program size and scale. Our work takes inspiration from Lopes and Ossher's study given that we 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.

9 Conclusions

This work presents a methodology to profile a large corpus of Alloy models and identify patterns in them using an ANTLR parser, and a combination of XPath and 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. In the "Characteristics of Models" section we assess the "surface-level" features of models. The "Modeling Practices" questions use complex patterns to investigate the use of the Alloy language constructs. We also compare the results obtained by examining the general corpus to the ones obtained by running our queries on a subset of expert models. More research questions and findings about our corpus of Alloy models can be found in Eid [41].

Some of our most interesting findings from this study, which result in actionable items, are:

- Although the results obtained from the general and expert corpora are fairly similar for most research

questions, expert models are far less likely to contain bad modeling practices.
- Scalars are used extensively in Alloy models.
- Alloy modelers use functions abundantly, presenting an opportunity for optimizations in analysis.
- The length of an Alloy model is correlated to the number of formulas rather than the number of sets.
- The ordering module is commonly used, but many places integers are being used by modelers to represent an ordered set.
- The vast majority of formulas are written in the relational calculus style.
- Expert modelers have a clear preference for relational calculus over the other formula writing styles.
- The set hierarchy typically used in Alloy is wide and partitioned meaning that tool developers may be able to create optimizations that address this particular structure of set hierarchy.
- Half of all Alloy queries use the default scopes for sets.
- Expert modelers tend to explicitly set scopes in commands and take advantage of scope derivation.

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. Future work can evaluate the models more qualitatively, perhaps looking for common modeling idioms and patterns, which could lead to tools that provide warnings (*e.g.,* Lint [49] for Alloy) and guidance for modelers.

Our work pioneers the direction of studying the use of declarative modeling languages. Our long-term goal is to better understand the best practices and value of declarative modeling languages in software engineering.

Acknowledgments

We thank Amin Bandali, Tamjid Hossain, Joseph Poremba, and Khadija Tariq, for discussions regarding this study. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Transactions on Software Engineering Methodology*, vol. 11, no. 2, pp. 256–290, 2002.

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

[3] J. Abrial, *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[4] ——, *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[5] J. M. Spivey, *Z Notation - a reference manual*, 2nd ed. Prentice Hall, 1992.

[6] C. B. Jones, *Systematic software development using VDM*, 2nd ed. Prentice Hall, 1991.

[7] E. Börger and R. F. Stärk, *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[8] P. Zave, "Using Lightweight Modeling to Understand Chord," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, pp. 49–57, 2012.

[9] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon Web Services Uses Formal Methods," *Communications of the ACM*, vol. 58, no. 4, pp. 66–73, 2015.

[10] N. Huynh, M. Frappier, H. Pooda, A. Mammar, and R. Laleau, "SGAC: A multi-layered access control model with conflict resolution strategy," *Computer Journal*, vol. 62, no. 12, pp. 1707–1733, 2019.

[11] D. Jackson, *Software abstractions: logic, language, and analysis*, 2nd ed. Cambridge, Mass: MIT Press, 2012.

[12] H. Wayne, *Practical TLA+: Planning Driven Development*. Apress, 2018.

[13] (2021) ABZ 2021 – 8th International Conference on Rigorous State Based Methods. [Online]. Available: https://abz2021.uni-ulm.de

[14] F. Boniol and V. Wiels, "The landing gear system case study," in *International Conference on Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, 2014, pp. 1–18.

[15] D. de Azevedo Oliveira and M. Frappier, "Verifying SGAC access control policies: A comparison of ProB, Alloy and Z3," in *Rigorous State-Based Methods*. Springer, 2020, pp. 223–229.

[16] A. Sree-Kumar, E. Planas, and R. Clarisó, "Analysis of feature models using Alloy: A survey," *Electronic Proceedings in Theoretical Computer Science*, vol. 206, pp. 46–60, 03 2016.

[17] A. Sullivan, K. Wang, S. Khurshid, and D. Marinov, "Evaluating state modeling techniques in Alloy," in *Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, September 11-13, 2017*, 2017.

[18] T. Nelson. (2022) Logic for Systems. Brown University. [Online]. Available: https://csci1710.github.io/2022/

[19] A. Cunha. (2021) Métodos Formais de Programação. University of Minho. [Online]. Available: https://haslab.github.io/MFP/

[20] S. Nakajima, "Using Alloy in introductory courses of formal methods," in *Structured Object-Oriented Formal Language and Method*. Springer International Publishing, 2015, pp. 97–110.

[21] J. Noble, D. J. Pearce, and L. Groves, "Introducing Alloy in a software modelling course," in *Formal Methods in Computer Science Education Workshop*, 2008.

[22] T. Ball and B. Zorn, "Teach foundational language principles," *Communications of the ACM*, vol. 58, no. 5, pp. 30–31, May 2015.

[23] L. C. Briand and J. Wüst, "Empirical studies of quality models in object-oriented systems," ser. Advances in Computers, M. V. Zelkowitz, Ed. Elsevier, 2002, vol. 56, pp. 97–166.

[24] J. Wüst. (2021) SDMetrics. [Online]. Available: https://www.sdmetrics.com/

[25] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of systems architecture*, vol. 57, no. 3, pp. 294–313, 2011.

[26] (2021) Welcome to Alloytools. [Online]. Available: https://alloytools.discourse.group

[27] (2021) Questions tagged [alloy]. [Online]. Available: https://stackoverflow.com/tags/alloy

[28] S. Farheen, N. A. Day, A. Vakili, and A. Abbassi, "Transitive-closure-based model checking in Alloy," *Journal of Software and Systems Modelling*, vol. 19, pp. 721–740, 2020.

[29] Amin Bandali, "A comprehensive study of declarative modelling languages," MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2020.

[30] Khadija Tariq, "Linking Alloy with SMT-based finite model finding," MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2021.

[31] D. Jackson, "Alloy: A language and tool for exploring software designs," *Communications of the ACM*, vol. 62, no. 9, pp. 66–76, Sep. 2019.

[32] (2021) org.alloytools.alloy. [Online]. Available: https://alloytools.org

[33] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 632–647.

[34] R. Yan, A. Bandali, and N. Day. (2021) Catalyst. [Online]. Available: https://github.com/WatForm/alloy-model-sets

[35] E. Eid. (2022) Static Profiling of Alloy Models. [Online]. Available: https://github.com/WatForm/static-profiling-of-alloy-models

[36] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Pragmatic Bookshelf, 2013.

[37] (2018, Mar.) 5.0.0 change list. [Online]. Available: https://github.com/AlloyTools/org.alloytools.alloy/wiki/5.0.0-Change-List

[38] (2021) The R Project for Statistical Computing. [Online]. Available: https://www.r-project.org

[39] (2021, Jun) What does it mean to "run" a function in Alloy? [Online]. Available: https://stackoverflow.com/questions/68146285/what-does-it-mean-to-run-a-function-in-alloy

[40] (2021, Jun) What is the difference between assertions and unparameterized predicates in Alloy? [Online]. Available: https://stackoverflow.com/questions/68076177/what-is-the-difference-between-assertions-and-unparameterized-predicates-in-allo

[41] E. Eid, "Profiling Alloy models," MMath thesis, University of Waterloo, David R. Cheriton School of Computer Science, 2021.

[42] T. Hossain and N. A. Day, "Dash+: Extending Alloy with hierarchical states and replicated processes for modelling transition systems," in *International Workshop on Model-Driven Requirements Engineering (MoDRE) @ IEEE International Requirements Engineering Conference (RE)*. IEEE, 2021.

[43] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham, "Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design)," *Object Oriented Systems*, vol. 3, pp. 143–158, 1996.

[44] (2013, Jun.) The util/ordering module and ordered subsignatures. [Online]. Available: https://stackoverflow.com/questions/17308778/the-util-ordering-module-and-ordered-subsignatures

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

[46] F. Erata. (2018) Alloy/kodkod benchmarks. [Online]. Available: https://tinyurl.com/alloy-benchmarks

[47] G. Zheng, H. Bagheri, G. Rothermel, and J. Wang, "Platinum: Reusing constraint solutions in bounded analysis of relational logic," in *International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2020, pp. 29–52.

[48] C. V. Lopes and J. Ossher, "How scale affects structure in java programs," *SIGPLAN notices*, vol. 50, no. 10, pp. 675–694, 2015.

[49] S. C. Johnson, "Lint, a C Program Checker," 1978.

Elias Eid received the MMath degree at the University of Waterloo in 2021. He received a BSc in Computer Science from Minnesota State University Moorhead in 2019. His research interests include declarative modeling languages, static analysis and empirical studies.

Nancy A. Day is an Associate Professor in the David R. Cheriton School of Computer Science at the University of Waterloo in Canada. She received her PhD and MSc degrees at the University of British Columbia and her BSc at the University of Western Ontario. Her research interests include modelling languages and model-driven engineering, formal specification and automated analysis, and verification methods to ensure the safety of software-intensive systems. She is a member of the IEEE Computer Society.