A Language for Feature-Oriented Security Policies using SMT

Silicon Valley Project Center
March 1, 2019

Submitted by:

Nathan Drewniak, *ndrewniak@wpi.edu*

Jeremy Hoffman, *jmhoffman@wpi.edu*

Paul Shingleton, *pcshingleton@wpi.edu*

Submitted to:

Project Advisor: Mark Claypool, *claypool@cs.wpi.edu*

Shape Security Advisor: Michael Ficarra *mficarra@shapesecurity.com*

Shape Security Co-Advisor: Tim Disney *tdisney@shapesecurity.com*

Worcester Polytechnic Institute

This MQP report is submitted in partial fulfillment of the degree requirement of Worcester Polytechnic Institute. The views and opinions expressed herein are those of the authors and do not necessarily reflect the positions or opinions of Worcester Polytechnic Institute.

Abstract

Shape Security, a cybersecurity startup, employs a reverse proxy server system named Pegasus to protect their customers' network traffic against attacks. Pegasus is configured by software in a feature-oriented paradigm, composing components of domain-specific code to tailor security policies for customers. Since Shape's composition system has no developer-written way to enforce constraints between components, creating valid compositions is difficult. Our project addresses this issue by enabling a means for predicates to be written for components by the developers. This allows the use of programmable first order logic to validate that all components in a given policy satisfy the features' predicates. The result is a new language which tests facts using logic to validate policies.

Acknowledgements

We would first like to thank our sponsor, Shape Security, for this fantastic opportunity to contribute to one of their internal technologies. We would like to thank our entire team at Shape, specifically our mentors, Michael Ficarra and Tim Disney, who have been incredibly helpful in guiding us throughout our project. Finally, we would like to thank our MQP advisor, Mark Claypool, for his help and guidance throughout the entire process and for giving us this amazing opportunity.

Table Of Contents

Abstract 2

Acknowledgements 3

Table Of Contents 4

List of Figures 7

1. Introduction 8

2. Background 10
- 2.1 Shape Security 10
  - 2.1.1 Pegasus and Policy Composer 11
    - 2.1.1.1 Feature Oriented Programming 11
    - 2.1.1.2 Pegasus 11
    - 2.1.1.3 Policy Composer 12
  - 2.1.2 DEX 15
  - 2.1.3 Athena 16
- 2.2 External Technology 16
  - 2.2.1 ANTLR 16
  - 2.2.2 Z3 16

3. Methodology 19
- 3.1 Project Requirements 19
- 3.2 Design of Language 20
- 3.3 Creation of Grammar 21
- 3.4 Creation of Parser 22
- 3.5 Creation of Z3 Code 22
- 3.6 Creation of Compiler 23
- 3.7 Testing 23

4. Implementation 25
- 4.1 Language design 25
- 4.2 Parser Design 28
  - 4.2.1 AST Nodes Class Structure and Design 28
  - 4.2.2 Lexer Implementation and Design 30
  - 4.2.3 Parser Implementation and Design 32

4.3 Semantic Analysis Design 35
4.3.1 Predicate checking 35
4.3.2 Formula checking 37
4.3.2.1 Checking Quantifiers 40
4.3.2.2 Checking Predicates 41
4.3.2.3 Checking Comparative Formulas 41
4.3.3 The Need for a Second Traverser 42
4.4 Integration with Z3 42
4.4.1 Translating to Z3 Code 43
4.4.2 Running the Z3 Code 45
4.4.3 Reporting Feedback from Z3 46
4.4.4 Ambiguity with Z3 46
4.5 Compiler Design 47
4.6 Testing 49
4.6.1 Lexer testing 49
4.6.2 Parser testing 50
4.6.3 Compiler Testing 51
5. Conclusion **53**
6. Future Work **55**
6.1 PCS Performance Optimization 55
6.2 UI Mockups for Integration 55
6.2.1 Mockup For Policy Composer Integration 56
6.2.2 Mockup For Error Reporting 57
7. References **59**
Appendix A: UML Diagrams **60**
Appendix B: Parsing Test Cases with ANTLR **61**
Appendix C: Z3 Proof when Unsatisfiable **68**

List of Figures

Figure 1: Pegasus Policy Creation overview 11

Figure 2: Z3 Unknown Return Example 15

Figure 3: AST Nodes diagram 25

Figure 4: Lexer accessory classes 27

Figure 5: JFlex RegEx mapping 28

Figure 6: Parser UML 29

Figure 7: AST root node, binaryFormula 31

Figure 8: Local fields in PCS Compiler 33

Figure 9: UML for Visitor Pattern with Traverser 35

Figure 10: Z3 Code Generation Process 41

Figure 11: UML for PCSCompiler 44

Figure 12: Lexer Test Cases for Expected Token Types 46

Figure 13: Parsing Test Cases 47

Figure 14: Testing Factual Predicates 49

Figure 15: Testing Formulas 50

1. Introduction

As the number of people using the Internet approaches 4 billion [1], cybersecurity is becoming increasingly important to today's society. More data is being stored online than ever before, varying from hotel or airline reservations to healthcare and banking information. With personal information being accessed online more frequently, attacks with the intention of retrieving this information are increasing in frequency, as well.

In an effort to prevent these attacks, Shape Security, a cybersecurity startup located in Mountain View, California, identifies and blocks malicious behavior from their clients' network traffic without affecting legitimate users from accessing their information [2]. To help block attacks Shape Security uses software called Pegasus, a reverse proxy server system that consumes and sends out requests.

The Pegasus system distinguishes malicious traffic from the traffic of legitimate users by using policies tailored for each individual customer. Customer policies are composed from a collection of individual components and are created by using Shape's Policy Composer tool. Policy Composer composes policies in a feature-oriented paradigm by arranging and compiling components written in a domain specific language named DEX. When an attack occurs, Pegasus detects the malicious behavior using a DEX policy and blocks it, while simultaneously gathering information about the origin of the attack. This information allows Shape to improve the policy that blocked the attacker and improve their machine learning models, without blocking legitimate users.

However, the major issue with Policy Composer is when there is the addition of new custom constraints, written as annotations in DEX components. This requires implementation of additional language support.The user of Policy Composer is currently not able to specify any additional constraints that must be met in a policy without adding this additional language support.

The goal of our project is to develop a system that allows the Policy Composer users to add additional constraints to the policy. Our system, called Predicate-based Composer System (PCS), replaces the existing annotation system. With PCS, the component writer adds predicates (facts that are true about the specific component) in comments as part of the component. After a policy is composed, PCS allows a user to write first order logic statements to determine if the predicates written about each of the components in the composition can be made true by checking the satisfiability using Microsoft's solver Z3. The end result is PCS, a new system implementing a language which tests predicates using first order logic to validate policies.

The rest of this report is organized as follows: Chapter 2 discusses the background information necessary to understanding our project; Chapter 3 explains the choices made in designing PCS, including determining the project requirements; Chapter 4 details the process behind implementing PCS and integrating it with Z3; Chapter 5 summarizes our conclusions, and Chapter 6 gives recommendations as to how our project can be advanced and maintained in the future.

2. Background

The sections that follow will describe Shape’s products that influenced our project and the technologies that were utilized to complete our project.

2.1 Shape Security

This section gives an overview of Shape’s terminology and the internal architecture that influenced our project.

2.1.1 Pegasus and Policy Composer

Pegasus and Policy Composer follow a feature oriented paradigm. Before going into detail about Pegasus and Policy Composer, we will define the concept of feature oriented programming.

2.1.1.1 Feature Oriented Programming

Feature Oriented Programming is a programming paradigm that focuses on developing features of a system, one feature at a time, until the desired system has been created [3]. A feature is a piece of system functionality that a user can identify. Different features require different capabilities, and different tools require different capabilities. A key goal is to allow third parties to add new features to existing products without modifying existing code. Although having many small components makes it

possible to assign the least authority to each one, it over-burdens the programmer having to link each one.

2.1.1.2 Pegasus

Pegasus is a scriptable reverse proxy server that is responsible for detecting and eliminating bot traffic and malicious behavior between client endpoints and their servers. Pegasus processes both the pre and post network traffic to track down information in the event that malicious behavior is detected. This data is captured through flags which can be used to block new attacks. Pegasus rules are specified through policy configurations, which are created using the Policy Composer.

2.1.1.3 Policy Composer

Policy Composer is the tool which allows Shape to create policies specific to their customer's needs and requirements. All policies are composed of many different *components*. Each component is a specific group of customizable configurations that helps filter network traffic. All of these components are written in an internal domain-specific language (DSL) called DEX.

Policy development consists of organizing a feature into a valid arrangement, providing the configuration arguments required by those features, and compiling the arrangement into a Pegasus policy. The Policy Composer is designed to achieve the following primary goals:

1. To develop and maintain Pegasus policies as a list of domain-specific features, offering an abstract layer in which policies can be composed and maintained by domain experts.
2. To build a library of policy feature components, which may be reused to create new policies. Feature components may be developed, tested and altered in isolation from other features.
3. To provide a simplified configuration layer for Pegasus policies through which policies may be configured by a domain expert who is not necessarily familiar with DEX or the implementation of Pegasus policies in DEX.

Figure 1 below outlines the steps necessary to create a policy with Pegasus. First, custom features are combined with standard features, loaded from a standard feature library. These are assembled into a policy feature composition. The features are arranged in this composition based on configuration parameters. A Pegasus Policy is then generated from this resulting composition. Our project will help aid this process by allowing more robust configuration parameters to be applied to the compositions.

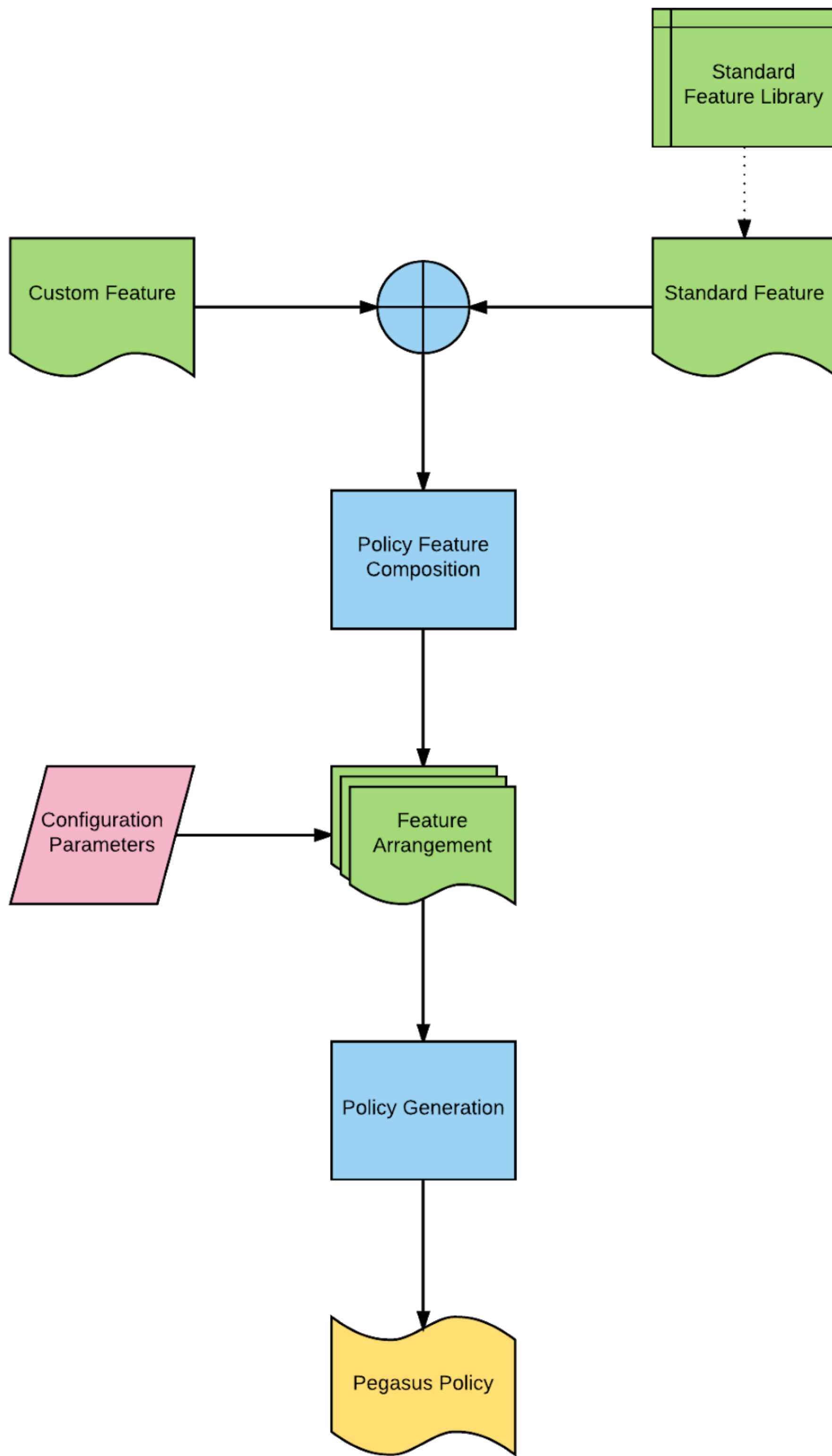


Figure 1: Pegasus Policy Creation overview

2.1.2 DEX

DEX is a DSL used to specify the behavior of the Pegasus reverse proxy server. DEX was developed at Shape and is used to create components of a policy for the composition of policies into features. Since there is currently no support for determining the interactions, intended or not, between different features, an extension to DEX must be created in order to pursue the idea of a sound composition algorithm. A problem with using DEX in its current state is the inability to determine the effects the feature would incur on top of the existing features. Thus, our project extends DEX to begin to address this problem.

DEX programs are executed within an input environment that provides input data and external computation. The result of executing a DEX program in a given environment is a collection of flags and reported values. Executing the same program in different environments may produce different results as the input data and external computation may be different.

There are two features of DEX that make it explicitly feature-oriented. The keyword 'super' provides access to identifiers defined in previous feature components in an arrangement of features. An arrangement of features is said to be valid if the data dependencies among the features are satisfied. Secondly, identifier names that start with underscore (_) are assumed to be local to the features in which they are defined.

2.1.3 Athena

Athena is a suite of tools which include a parser, compiler, and interpreter for DEX. Composing a valid arrangement of features requires the DEX file to be parsed, compiled, and executed by Athena. As with most compilers, it has a lexer to scan the language for tokens, parses it into an AST, then eventually compiles.

2.2 External Technology

In this section, we will discuss external technologies that we used to help in the design and implementation phases of our language.

2.2.1 ANTLR

ANTLR (Another Tool for Language Recognition), is a tool that can automatically generate a parser from lexer and parser rules, specified in regular expressions and context-free grammars. ANTLR is especially useful since it can graphically show the parse trees from input text, which aids in ensuring the correct structure of a tree from a given grammar. This allows for the easy modification of the grammar, without having to refactor the parser by hand. These trees can be viewed in Appendix B. We used ANTLR to visualize the ASTs when designing our grammar.

2.2.2 Z3

SAT solvers attempt to determine if there exists an interpretation that satisfies a given boolean formula. They try to find a solution such that a boolean formula can

evaluate to being true. As of 2007, SAT-algorithms have been able to solve problems consisting of thousands of variables and millions of symbols, though not in polynomial time due to their NP-Complete status [4].

Modern SAT solvers have similar features, which include watched literals, learning mechanisms, deterministic and randomized restart strategies, cause deletion mechanisms, and smart static and dynamic branching heuristics. Since the SAT problem is NP-Complete, the additions of complexities within the formulae cause the runtime efficiency to increase non-polynomially.

One popular and efficient solver is Z3, a modern Satisfiability Modulo Theory (SMT) prover developed by Microsoft Researchers [5,6]. Z3 takes logical formulas and expressions and assembles them into a single composition. Z3 then determines the satisfiability of the entire composition.

There are three possible returns from checking the satisfiability of a Z3 composition. The first return option is “sat.” *Sat* is returned if there exists a model that is satisfied for every defined formula in the composition, thus satisfiable. Next, “unsat” is returned if there is no possible model that can satisfy every formula, thus unsatisfiable. The final possible return is “unknown.” While rare to get unknown as a return, it occurs if Z3 cannot determine the behavior of a formula present in the composition. This is found when Z3 cannot determine the satisfiability of a formula.

```
(declare-const a Int)
(assert (> (* a a) 3))
(check-sat)
(get-model)

(echo "Z3 does not always find solutions to non-linear problems")
(declare-const b Real)
(declare-const c Real)
(assert (= (+ (* b b b) (* b c)) 3.0))
(check-sat)
```

Figure 2: Z3 Unknown Return Example

Figure 2 shows an example when Z3 returns unknown. The first “(check-sat)” returns *sat* since Z3 can determine a valid solution. However, the second “(check-sat)” results in *unknown*. Z3 cannot determine if there is a solution for both of the assertions. The satisfiability can be checked at any portion of the composition, which is useful for determining which formula caused an undesirable return. We use Z3 for determining if a solution is available to satisfy facts and first order logic statements made about a composition.

3. Methodology

In the sections that follow, we outline the planning process we took in completing our project and creating Predicate-based Composer System (PCS). This includes our project requirements, the design of our new language, the grammar associated with that new language, the creation of our parser, creation of Z3 code, the creation of our compiler and testing.

3.1 Project Requirements

In order to discover the most important and useful features to add to DEX, we consulted with many different teams to understand their perspectives and determine what would be the most beneficial. Currently, there are annotated metadata before modules, components, or bindings in a DEX file. These metadata store specific information about their respective part of the DEX file.

One of the major issues with the current system is that all the annotated metadata is hard coded, meaning personalized or custom annotations cannot be added. If there were a way that custom annotations could be allowed, this would make it easier for both the writers of DEX policy components and the Policy Composer team.

The writers of the components would benefit from custom annotations since they would be able to add certain annotations to components that would have an internal meaning to their team. For example, if they wanted a few specific components to all be

part of one group, they would be able to by labeling them and specifying how components with that label may interact.

Once these components are written with the custom annotations, they would then be used by the Policy Composer team. The policy composers would benefit from these custom comments since there would be newer, more specific rules in place about which composition of components can make a valid policy. If there were a group of components where only one from the group could be chosen, it would be much easier for the Policy Composer team to identify that requirement and be able the pick the component from that group that best suits their needs for that specific policy, making the entire policy composing process easier.

3.2 Design of Language

Once we determined the specific requirements needed, we were able to design a solution. Since the current system used an annotation system and all the annotations were hard coded, using a predicate system of logical facts would be a better approach. Our language, composed of these predicates, would be parsed into a series of First Order Logic statements. We visualized many examples of predicate statements and what their corresponding logical statements would be.

As an example, one component may have the annotation "@Requires [*anotherComponent*]," which states that *anotherComponent* needs to be included in the composition of a policy if the annotated component is also part of the policy. The first order logic statement for this would translate to $\forall x,y.\ Component\ (x) \wedge Requires(y) \rightarrow$

Component(y). In this case, *Component(x)* evaluates whether *x* is a component in the current policy, and similarly, *Requires(y)* evaluates whether a component *y* is required. This statement would be evaluated, and if it evaluates to *false*, then that specific policy composition would fail, since it requires the second component to also be present in the composition.

After testing of our language, we were able to discover some unnecessary requirements that were present originally. PCS initially required a component *x* to also have a corresponding *Component(x)* for defining it as a component and usage in first order logic statements. Through type checking, we were able to remove the need to this. Instead of having to say explicitly in Z3 that *x* was a component as before, PCS was able to determine that it was a component based on its type.

3.3 Creation of Grammar

After gathering requirements for our language, we created a grammar to define it. In order to test our grammar, we needed to be able to parse it and see the AST that it produces. To make this easier for us, we used ANTLR, an automatic top-down parser generator. In these planning stages, we were able to rapidly prototype a grammar and test the generated parser for correctness. To ensure the quality of our grammar, we wrote many tests using valid first order logic statements, which can be viewed in Appendix B.

3.4 Creation of Parser

Once we determined the requirements for our grammar, we created a parser for PCS. The parser needs to identify the required tokens from the DEX comments to correctly parse out the desired information. All the AST Nodes evaluate to a *Formula* or a *Term*. Formulas are the predicates that are written directly in the DEX comments. Terms are the parts of the predicates that would eventually evaluate to a data type. The parser for PCS was written in Java.

3.5 Creation of Z3 Code

PCS uses Z3 as the tool to evaluate first order logic. Z3 has a Java API available for use, but there is not much documentation for using this API. Through manual experimentation, we found that the user can create Z3 code by calling the various functions for creating sorts, constants, and functions as needed for the composition. The downside for this API is that it is difficult to create formulas with quantifications, and all the first order logic we use contains at least one, and often multiple, quantifications. For our use, we would need to create the inner formula that was being quantified over first, then quantify over it. This proved to be difficult when introducing multiple and nested quantifications.

The other option we pursued was to manually generate Z3 code through code generation. Using the style that was available on Rise4Fun, Microsoft's website for

learning Z3, PCS could generate sample logic examples and test them. Using the Z3 Java API, PCS could pass this code as a string to create a solver based on the code. Then, PCS could evaluate the satisfiability of the solver. We used this method when integrating Z3 into PCS.

3.6 Creation of Compiler

Once we designed the parser and determined how we would integrate Z3, we developed a compiler for PCS. This compiler contained a list of the predicates being parsed from the DEX comments and compiled them to Z3 code. The compiler also needed to account for the various differences that Z3 code provided, such as defining sorts, as well as to accommodate to the way Z3 code is structured. This accomodation required declaration of predicate signatures as Z3 functions along with adding in the usages of the predicates as assertions. After generating the Z3 code, the PCS compiler attempts compilation using the Z3 Java API and returns the satisfiability to the user, as a boolean value, for the specific first order logic statement being tested.

3.7 Testing

Due to the unique nature of Z3 and its interaction with the parser and compiler, testing needed to be robust. We tested the parser and the compiler for PCS as its development progressed, verifying outcomes or modifying the code based on the results.

The parser was tested to verify the ASTs were structured properly and only parsed valid tokens. We ensured these ASTs worked according to the precedence and associativity for all the operators included in our language.

Next, the compiler was tested extensively to ensure that errors were thrown when expected to correctly validate predicates, and translate to Z3 code. The compiler was also tested to cover all the errors that Z3 could provide, so that any Z3 error would be due to an *unknown* return. All testing was done through a combination of JUnit tests and visual tests to confirm the structure of the parser trees.

4. Implementation

The following sections detail the implementation of our API for compiling our language, Predicate-based Composer System (PCS), which also alludes to Policy Composer with the first two letters.

4.1 Language design

This section describes the language we designed, as well as the design decisions behind it.

4.1.1 Syntax

The syntax for our language (which can be seen in detail below) was designed to make writing first order logic simple. The syntax also is designed such that anything that is written must be a *formula*, or something that results to an assertion of a boolean formula or predicate, so that semantic analysis does not have to check for errors caused by inputs that are not directly translatable to Z3. The syntax also implies that the language is restricting input to only binary expressions; everything formula-wise is connected to another formula by connective operators (implication, biconditional, conjunction, disjunction).

$$formula \rightarrow [\ \{FORALL \mid EXISTS\}\ V^{+}.\ ]\ formula\ [\ connective\ formula\ ]^{*}$$

$$\mid\ \tilde{}formula$$

$$\mid\ '('\ formula\ ')'$$

$$\mid\ predicate$$

$$\mid\ comparative$$

$$V \rightarrow (a-z)(a-zA-Z0-9_)^{*}$$

$$connective \rightarrow [=>\ \ <=>\ \ AND\ OR]$$

$$predicate \rightarrow pid\ tuple$$

$$\mid bindingLevel\ :\ \#id$$

$$bindingLevel \rightarrow \{thisComponent \mid thisModule \mid thisBinding\}$$

$$id \rightarrow V$$

$$pid \rightarrow\ (A-Z)(a-zA-Z0-9_)^{*}$$

$$tuple \rightarrow '('\ term\ [,\ term]^{*}\ ')'$$

$$term \rightarrow string$$

$$\mid num$$

$$\mid thisComponent$$

$$\mid thisModule$$

$$\mid thisBinding$$

$$\mid true$$

$$\mid false$$

$$\mid \#identifier$$

$$\mid tuple$$

$$identifier \rightarrow V$$

$$comparative \rightarrow term\ op\ term$$

$$op \rightarrow (<\ \ >\ \ <=\ \ >=\ \ =\ \ !=)$$

Notes: parenthesis represent sets, whereas curly brackets represent grouping. Brackets represent optionality. ‘AND’ and ‘OR’ are left-associative and have lower precedence than ‘=>’ and ‘<=>’ which are right-associative.

4.1.2 Predicates

Predicates, written in the comments of DEX files, can be used to assert facts about the AST Node they are attached to. The three levels of interest by which expressing facts about and between them would be useful are the following: Components in Policy Composer; Modules (analogous to a Class in Java); and (variable) Bindings in DEX. These varying *levels* of interaction have been implemented into PCS as a necessary means to record facts about them. The keywords "thisComponent", "thisModule", and "thisBinding" represent those levels, henceforth referred to as *Binding Levels*. A predicate parsed from DEX AST Nodes must use one of the binding level keywords in order for logical quantification to be able to distinguish between the three domains of discourse.

Predicates are also implemented to record facts about the specific *level instance* (a specific component, module, or binding) they are attached to. Our language offers *terms:* integers, strings, booleans, and specific references to other level instances, called identifiers (prepended with a '#'). Using these primitive data types as well as references to level instances allow a basis for allowing the statement of facts about level instances.

4.1.3 First Order Logic Formulas

With the ability to state facts about level instances, being able to reason about them is required to be able to redefine validity in compositions. Quantification, via universal and existential operators (FORALL, EXISTS), as well as the *variable* used to

quantify over, in addition to the connective operators, and comparative operations, allow a robust way to reason about the predicates in the features. By having test cases written in first order logic and translating them to our language, we were able to create this simplified language that could encapsulate the functionality required. Below is an example of a formula and its respective PCS equivalent.

$$\exists y.\ \forall x.\ Name(x,y) \land \forall z.\ \forall a.\ Name(z,a) \Rightarrow a = y \land y = \text{“}findsHackers\text{”}$$

EXISTS y. FORALL x. Name(x,y) AND FORALL z. FORALL a.

Name(z,a) => a == y AND y = “findsHackers”

4.2 Parser Design

The following sections detail the design of the parser that was implemented.

4.2.1 AST Nodes Class Structure and Design

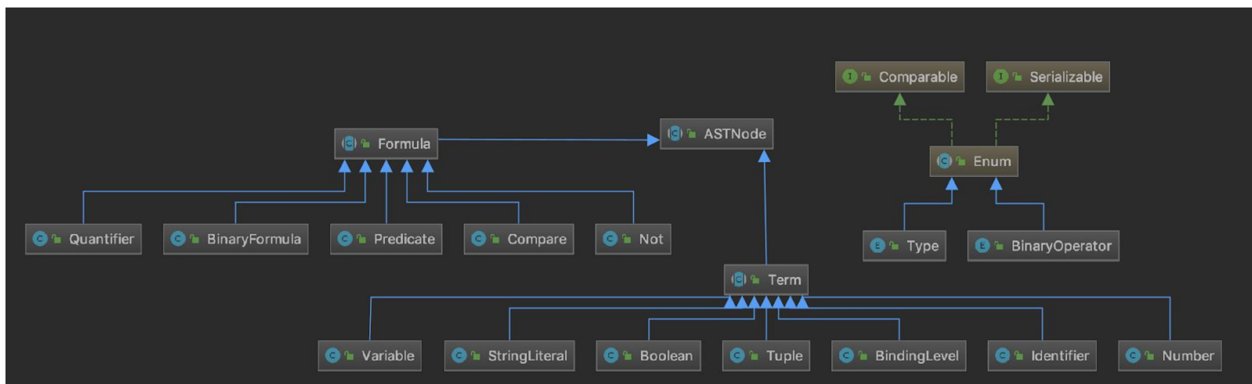


Figure 3: AST Nodes diagram (see Appendix A for UML)

The AST Node classes were designed around *Formula* and *Term* due to the effect that each has; a formula can be asserted as being true, whereas terms resolve to a data type. The AST Nodes diagram can be seen in Figure 3 above, where Appendix A shows the complete UML diagram that includes the method descriptors for each member. In designing the parser, a functional approach was used considering the nature of programming languages being deterministic; note the deterministic typing of children for each node - a *BinaryFormula* has exactly two *Formula* as children, with one *BinaryOperator* enum. Also noteworthy is the usage of a *Type* enum - for terms with known types, the types are set in the constructor to their respective values, whereas terms like *Identifiers* are typed in the later stages of the compiler. Finally, PCS uses Shape Security's own *Functional Java* library [7] to further implement in a functional paradigm, using the types *ImmutableList<T>* to ensure immutability.

4.2.2 Lexer Implementation and Design

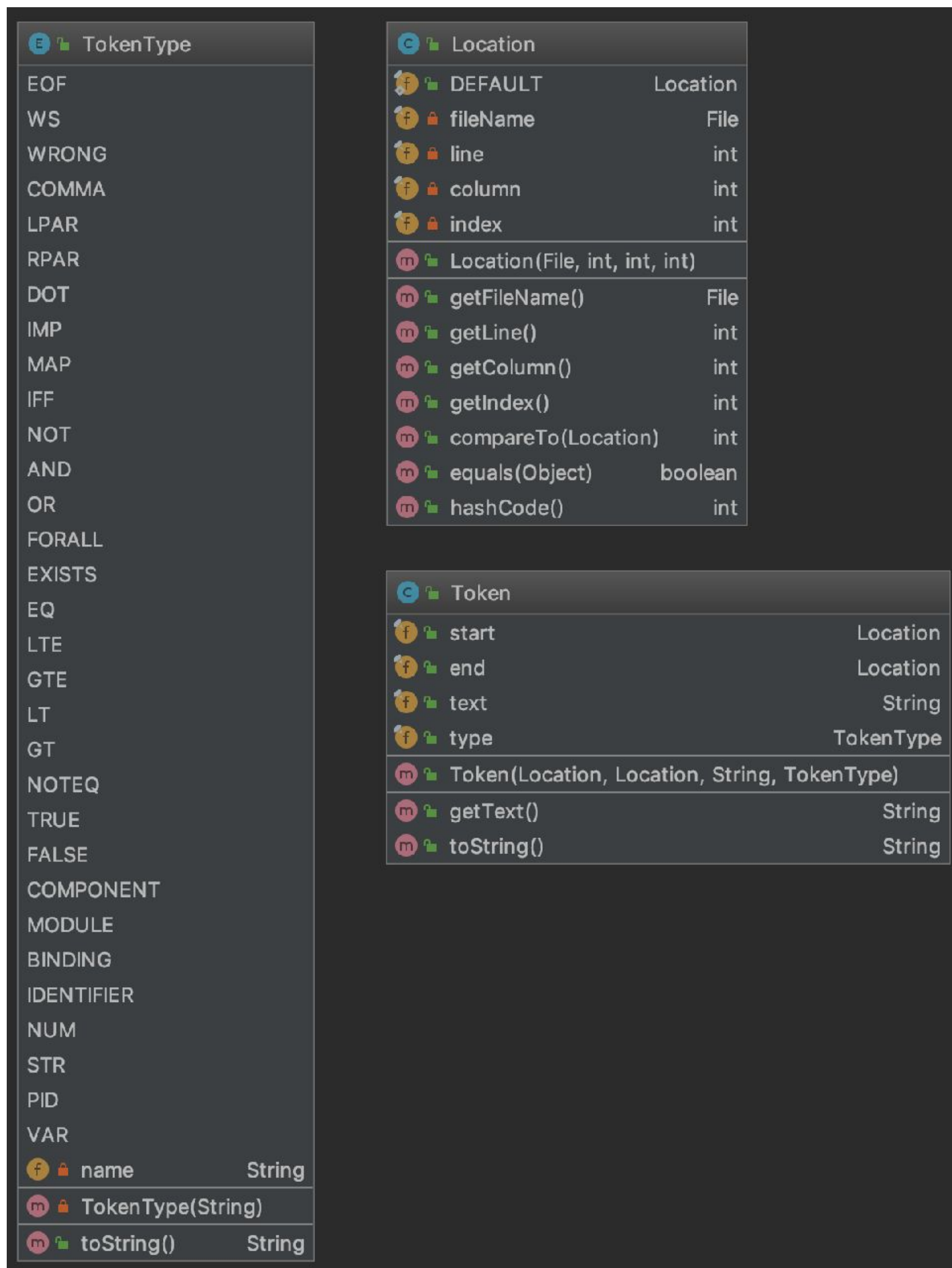


Figure 4: Lexer accessory classes

The lexer for the language was generated using JFlex, a library that allows the automatic generation of a lexer java class based on regular expressions and helper

class definitions [8]. It first converts the regular expressions into a non-deterministic finite automata, then converts it to a deterministic finite automata which is then used to capture input text and map those captured expressions to tokens. JFlex was used to map (Figure 5) the *TokenType* Enums from Figure 4 to their respective regular expressions. JFlex also records the location, in an x and y coordinate system, allowing the tracking of location to be translated in our own *Location* file, used later in error reporting. The lexer has a simple API which generates a list of tokens from an input string, used in the parser.

```
85   mVAR_START = [a-z]
86   mPID_START = [A-Z]
87   mID_PART = [a-zA-Z0-9_]
88   HASH = [#]
89
90   mPID = {mPID_START} {mID_PART}*
91   mVAR = {mVAR_START} {mID_PART}*
92   mID = {HASH} {mVAR_START}{mID_PART}*
93
94   %%
95
96   <YYINITIAL> {
97
98   {mWS}+                    {return WS;}
99
100  "`" [^`]* ("``"[^`]*)* "`"                          { return STR; }
101  "\"" ([^\"\\\n\r] | \\ [tvrnfb\\'\"])* "\""         { return STR; }
102  "\"\"\"" ([^\"]|\"[^\"]|\"\"[^\"])* "\"\"\""        { return STR; }
103
104        ","                 {return COMMA;}
105        "("                 {return LPAR;}
106        ")"                 {return RPAR;}
107        "."                 {return DOT;}
108        "=>"                {return IMP;}
109        "<=>"               {return IFF;}
110        "->"                {return MAP;}
111        "~"                 {return NOT;}
112        AND                 {return AND;}
113        OR                  {return OR;}
114        FORALL              {return FORALL;}
115        EXISTS              {return EXISTS;}
116        "=="                {return EQ;}
117        "<="                {return LTE;}
118        ">="                {return GTE;}
119        "<"                 {return LT;}
120        ">"                 {return GT;}
121        "!="                {return NOTEQ;}
122        true                {return TRUE;}
123        false               {return FALSE;}
124        thisComponent       {return COMPONENT;}
125        thisModule          {return MODULE;}
126        thisBinding         {return BINDING;}
127
128  {mPID}                    {return PID;}
129  {mVAR}                    {return VAR;}
130  {mID}                     {return IDENTIFIER;}
```

Figure 5: JFlex RegEx mapping

4.2.3 Parser Implementation and Design

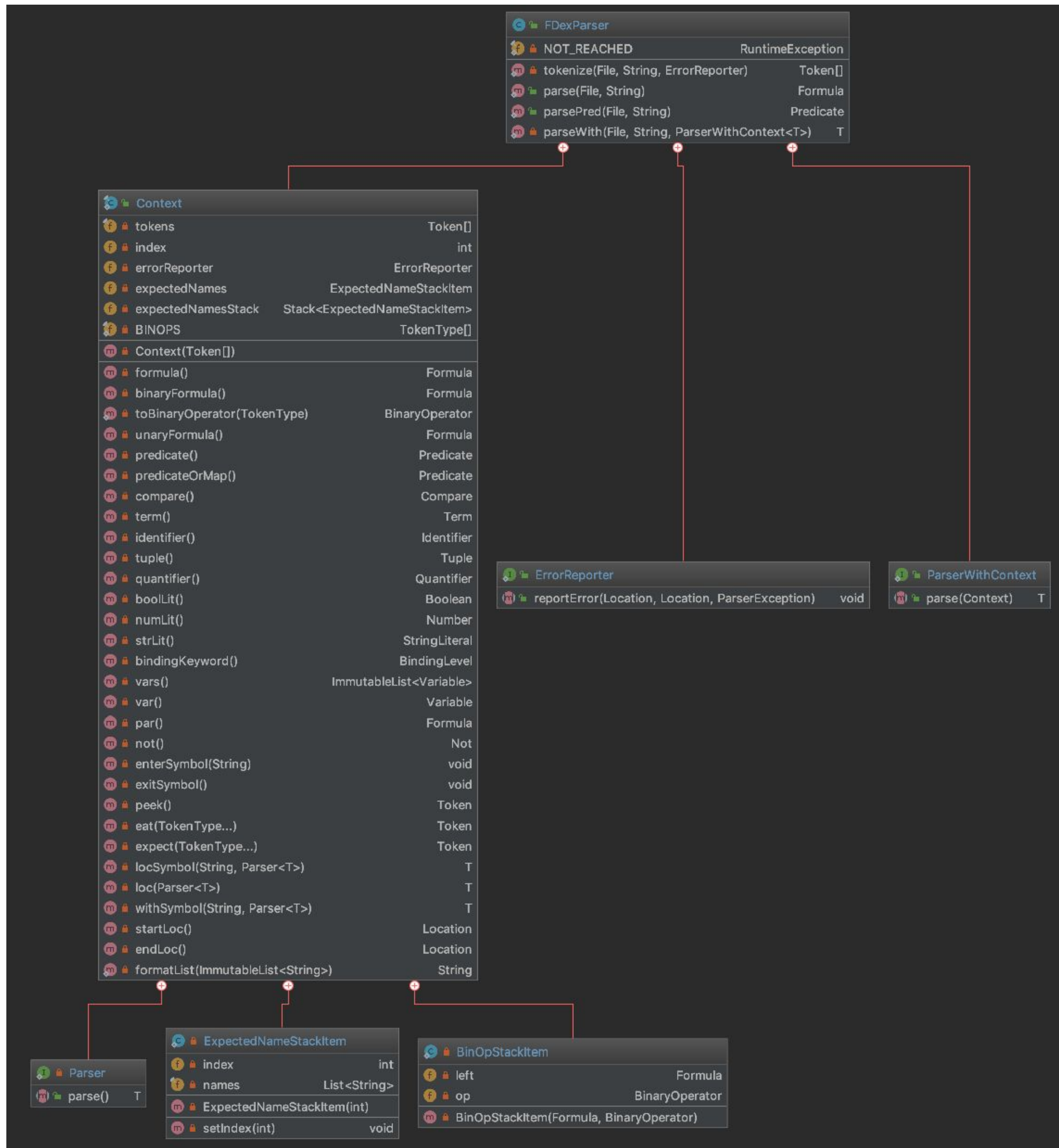


Figure 6: Parser UML

As shown in Figure 6, the parser, designed alongside the AST Nodes, also operates in a functional way; every outcome is deterministic and the resulting AST is immutable, with direct typed mapping for children. The API, being *parse* and *parsePred*, are designed to parse our language's formulas and predicates, respectively, given a file and a string.

The inner class *Context* is responsible for the methods involved in implementing the productions of the grammar, using a functional interface *Parser* which every production rule method implements. The interface is implemented in these methods by calling the functions *loc*, *locSymbol,* and *withSymbol* to wrap the insides of the functions with a call to *loc* etc., so that they may repeat the task of recording production rules and/or locations in a stack data structure system. Figure 7 below illustrates an example use case.

```
87      /**
88       * @return binaryFormula or unaryFormula
89       * Could also be singular expression.
90       * Shunting yard algorithm
91       * first(bf) = {PID, VAR, NUM, STR}
92       */
93      @Nonnull
94      private Formula binaryFormula() throws ParserException {
95          return this.locSymbol( name: "Binary Formula", () -> {
96              Formula lhs = this.unaryFormula();
97              Token opToken = eat(BINOPS);
98              if (opToken == null) {
99                  return lhs;
100             }
101             BinaryOperator op = toBinaryOperator(opToken.type);
102             Stack<BinOpStackItem> stackItems = new Stack<>();
103             stackItems.push(new BinOpStackItem(lhs, op));
104             lhs = this.unaryFormula();
105             while (true) {
106                 opToken = this.eat(BINOPS);
107                 op = opToken == null ? null : toBinaryOperator(opToken.type);
108                 if (op == null) {
109                     break;
110                 }
111                 while (!stackItems.isEmpty() &&
112                         ((op.isLeftAssociative() && (op.getPrecedence() <= stackItems.peek().op.getPrecedence())) ||
113                                 (!op.isLeftAssociative() && (op.getPrecedence() < stackItems.peek().op.getPrecedence())))) {
114                     BinOpStackItem item = stackItems.pop();
115                     lhs = new BinaryFormula(item.left, lhs, item.op);
116                 }
117                 stackItems.push(new BinOpStackItem(lhs, toBinaryOperator(opToken.type)));
118                 lhs = this.unaryFormula();
119             }
120             while (!stackItems.isEmpty()) {
121                 BinOpStackItem item = stackItems.pop();
122                 lhs = new BinaryFormula(item.left, lhs, item.op);
123             }
124             return lhs;
125         });
126     }
```

Figure 7: AST root node, binaryFormula

Notice the return on line 95 being a call to *locSymbol*, which manipulates the stack by adding the symbol “Binary Formula”, representing the production rule, to it.

The root of every *Formula* AST is created by the *binaryFormula* method. This method implements the Shunting Yard algorithm to parse a formula as a binary expression [9]; with the connective operators, a formula is just a binary expression where *AND* and *OR* being lower precedence with left-associativity and *IFF* and *IMP* being higher precedence with right-associativity. The call to *unaryFormula* attempts parsing individual formulas separated by these operators, being Predicates and comparative ones along with a logical *Not* and grouped formulas within parenthesis.

4.3 Semantic Analysis Design

The following sections detail the design of the multiple steps involved in semantic analysis, including usage safety, type checking, type inferencing, and variable binding management. All such functionalities are included in the same class for code generation, *PCSCompiler*.

4.3.1 Predicate checking

The set of all PCS predicates parsed from the DEX ASTs are analyzed first. The compiler is initialized with parsed predicates, in a *Map<T, ImmutableList<Predicate>>*. This format allows a fundamental separation of each level instance's PCS predicates, agnostic of the AST node mapped to it.

There are many different data structures used to store information about the ASTs given, shown in Figure 8. They store data that is discovered through multiple AST visits, and since the ASTs are immutable, they are never modified, and all knowledge discovered must be recorded.

```java
24        /**
25         * Given predicates.
26         */
27        @Nonnull
28        private final Map<T, ImmutableList<Predicate>> preds;
29
30        /**
31         * Each level has an ID mapped to it.
32         */
33        @Nonnull
34        private final Map<T, String> symbolsByNode;
35
36        /**
37         * 'Method' Descriptors of Predicates mapped to their PID. No overloading.
38         */
39        @Nonnull
40        private final Map<String, Predicate> predicateSignatures;
41
42        /**
43         * Map of identified and generated BindingLevels: Symbol -> Type
44         */
45        @Nonnull
46        private final Map<String, Type> identifiers;
47
48        /**
49         * Visitor ensures variable usage safety, variable type entry among quantifier scopes
50         */
51        @Nonnull
52        private final Traverser bindingVisitor;
53
54        @Nonnull
55        private final Traverser typeSetter;
56
57        @Nonnull
58        private final SymbolTable table;
59
60        @Nonnull
61        private final ArrayList<String> errors;
62
63        /**
64         * Storing all gen'd code in here
65         */
66        private final StringBuilder code;
```

Figure 8: Local fields in PCS Compiler, referenced in the following sections.

There are multiple steps that the predicates must go through to be passable for code generation. The format of each predicate must be consistent:

1. The first argument of each predicate must be a *BindingLevel*; arity must be > 0
2. The BindingLevel keyword may be used once and only once.

3. Each level instance must have its respective predicates use the *same type* of BindingLevel.
4. Variables are not allowed in the predicates parsed from the DEX ASTs.
5. Each AST in the map as generic T must have at least one Predicate as its value.
6. Using the ID mapping syntax can only be used once per level instance.

After this light checking is complete, PCS then must iterate through each level instance and determine if there was an explicit identifier mapping created for that level. If there is not, PCS generates a symbol to represent it, as Z3 will require it explicitly as a constant. PCS checks that each ID is unique between all level instances, and then place them into our *identifiers* map and *symbolsByNode* map.

Next, PCS must record the so-called method-descriptors of each predicate being used, as only their usage implies their existence. PCS populates our *predicateSignatures*, ensuring there are no other predicates with the same name that *are not* congruent to the existing ones.

Upon completion, PCS is able to say that the predicates are valid. Otherwise, an informative error, or series of them, are thrown for each violation, mentioning the specificity of the error and the name of the offending predicate(s).

4.3.2 Formula checking

In order for a formula to be checked, there must be an instance of *PCSCompiler*, for there can only be an instance for a successful check of given predicates (henceforth, *factual predicates)* by design (see Section 4.4 for the compiler design). This means that

there are now data structures for recording the factual predicate signatures, the variable type mappings, and the level instances mapped to an identifier or generated symbol. In addition to these structures, the two visitors *pre* and *post* are created to traverse formula ASTs, shown in Figure 9 below.

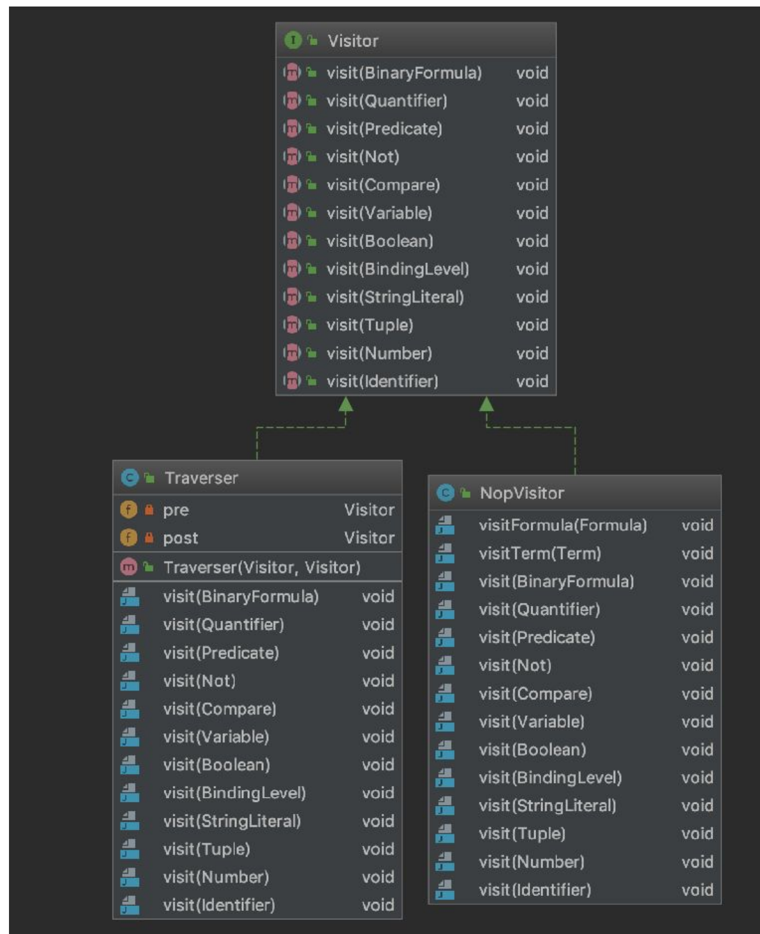


Figure 9: UML for Visitor Pattern with Traverser

The visitor pattern used to traverse the formulas is a variant on the standard visitor pattern - a *Traverser* class is used instead of having to implement every method in the interface. The Traverser takes in two visitors, *pre* and *post,* and visits a node by letting *pre* visit first, then the children would be visited by the traverser, finally *post*

would visit. The Traverser is used in the checking of formulas by visiting given formula ASTs after predicate checking, with the visitors being initialized as anonymous inner-classes of the *NopVisitor* type. NopVisitor implements the Visitor interface but has entirely empty implementations. The implications of this pattern is that PCS can specify how to visit a particular node before *and* after visiting children, without having to implement any other of the visitor methods.

The checking of formulas has multiple stages and requires robust techniques to deal with quantification. There are many complications that arise with expressing quantification, especially in the context of translating to code. The first complication that arises is chained quantification; the expression of multiple domains.

Take, for example, the following formula:

$$FORALL\ x.\ P(x)\ \Rightarrow\ (FORALL\ y.\ Q(y))$$

This formula's outer domain is $x \subseteq X$ whereas there is an inner domain $x \subseteq X \cup y \subseteq Y$, where X and Y represent the set of all variables with types x and y, respectively. This statement can be generalized inductively: In this formula, if we introduced a third quantifier within the inner domain, its domain would expand to include all the parent domains; for any given formula F, if F has a quantifier, there must exist a domain D in F for which all formulas in F are in scope, and similarly for every sub-formula. A variable referenced outside a scope where it was defined is free; a variable referenced within a scope where it was defined is bound. This problem introduces the need for a scope capturing system.

The next problem with quantification is typing. Examine the following FO formula:

$$\exists x\, y.\, P(x) \;\Rightarrow\; \exists\, z\, a.\, Q(x,y) \;\Rightarrow\; a = y \wedge z = x$$

The problem with translating this formula into a computerized system is that a type is required for each variable quantified over, so that such a system can discover the respective domains with inferring type itself, as is the case with Z3. This formula has variables *x* and *y* which never even get used until the next quantified formula, until PCS enters a new scope, and the type can only be inferred from the way the factual predicates have defined *Q(x,y)*.

To address these problems we introduce the Symbol Table as a means to ensure proper scoping techniques are used, as well as two Traversers to handle type inferencing. Together, they check the validity of formulas, the scoping, and typing to ensure proper code generation.

The following subsections describe what each visitor does for checking formulas. In all cases, PCS throws an error if a BindingLevel is used within a formula.

4.3.2.1 Checking Quantifiers

The first step in visiting quantifiers is entering each variable into the symbol table, mapping their name to their type. As the rest of the AST is visited, the type will resolve in the table if the formulas are proper. PCS “enters scope” in the table, which increments the index of position in the table. Upon completion, the second NopTraverser exits scope, decreasing.

4.3.2.2 Checking Predicates

There are myriad checks to be done for the predicates in a formula to assure they are well-formed. For each formula, PCS ensures the usage of every predicate, argument-type and arity-wise, is congruent to the factual predicates. PCS also ensures that every predicate referenced within a formula *has* a usage from the factual predicates. Finally, PCS also infers the types of variables having unknown types based on the way that they are being used in the factual predicates.

4.3.2.3 Checking Comparative Formulas

Checking a comparison between two terms is a dynamic task. Depending on which operator is being used, the types of both sides must either conform to a set of types, or be inferenced. If the operator is one of (<,>,<=,>=), then both sides must be numbers, or if they are of unknown type, are inferred to be a number. Next, if the operator is (==, !=), then PCS cannot directly infer type; in both cases, if one side of the operator is of a known and valid type while the other side is unknown, PCS can infer that type to be that of the known type. PCS must throw an error if the types cannot be inferred, or if they do not match.

At this stage, since all comparative operators with variables inside them are visited after a quantifier, the updating of a variable inside the symbol table must be done carefully. The table tries to resolve variables starting in the innermost scope first, working its way through parent scopes searching for a match. Ambiguities between

variables of the same name and different scopes have no solution, as a uniquely identifiable system for variables, as they are known to the compiler, has not been implemented.

4.3.3 The Need for a Second Traverser

Although the design allows two visits per node, a second Traverser is still needed. By allowing code generation to have the knowledge of the discovered types, we must either build another anonymous inner class within the compiler so that it has access to the knowledge from the local fields (the variable-type mappings), or we put that knowledge inside the AST nodes themselves and add a method for code generation in each node class. We decided on the latter since the nodes should have knowledge of their discovered type.

The second traverser sets the type of each variable, whose type was initially unknown, to the types discovered and recorded in the symbol table, additionally ensuring no free variables are allowed.

4.4 Integration with Z3

When introducing first order logic to PCS, we needed to figure out how PCS would evaluate and enforce the logic statements that users would make. Being an SMT solver, Z3 allows assertions made through predicates and the first order logic statements in one composition. PCS could then evaluate the satisfiability of this

composition and return it as a boolean value; true for *sat*, false for *unsat*, and throw an error in the rare instance that Z3 returns "unknown."

4.4.1 Translating to Z3 Code

Due to the unique nature of the Z3 language, PCS translates the content stripped from the DEX comments to Z3 code directly through code generation. Z3 requires knowledge of each variable's type using it in certain cases, such as declaring a function or asserting a quantified expression. This is not something PCS requires when the user is writing their first order logic statements or predicates. Instead, our design allows us to look ahead at how the variable will be used to determine the type it must be. This allowed us to generate the correct Z3 code needed for each code generation portion, while maintaining simplicity from the user perspective.

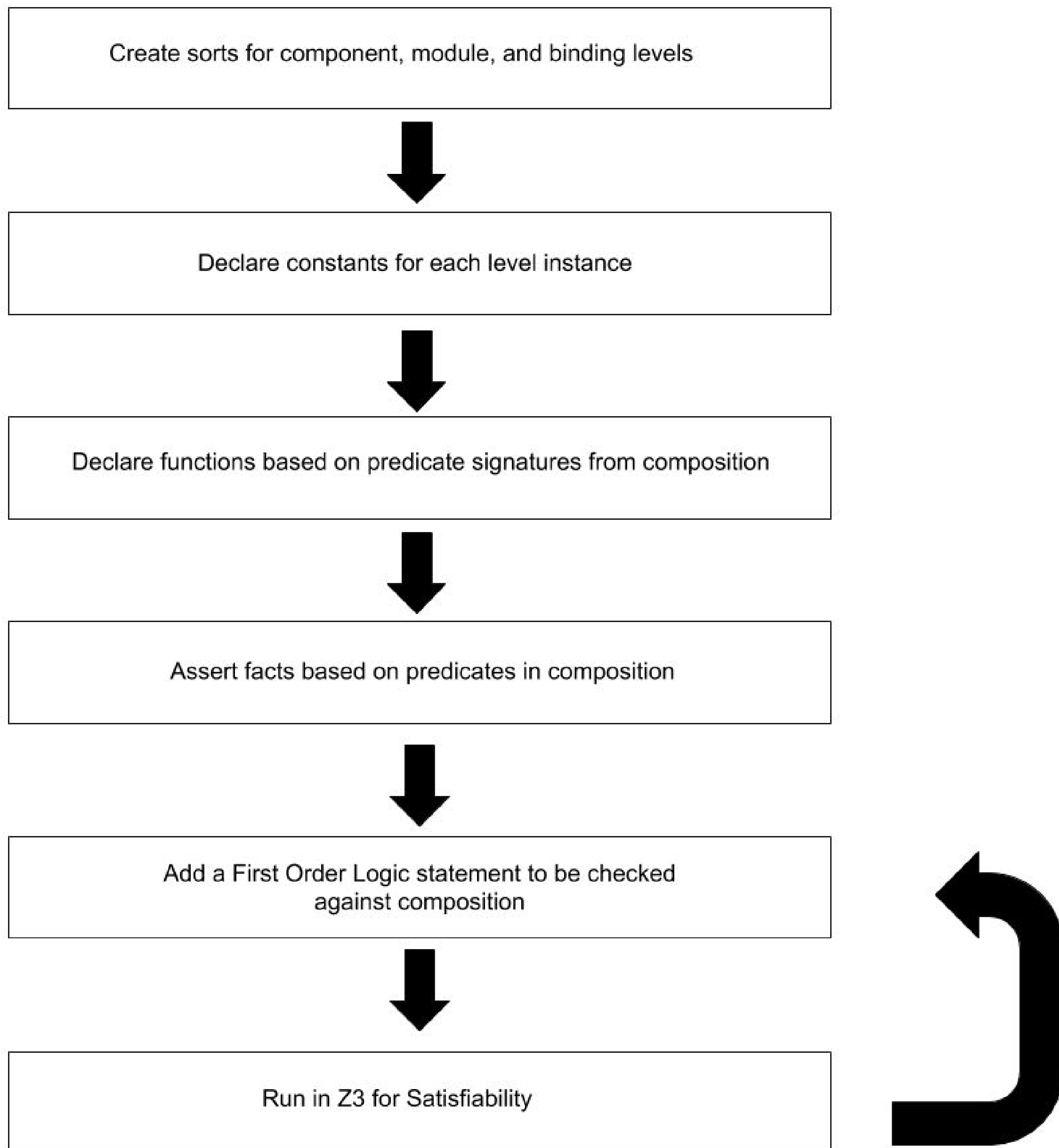


Figure 10: Z3 Code Generation Process

The code generation process can be seen in Figure 10 above. It begins by creating the custom variable types (called “sorts” in Z3) for the three binding levels. Next, PCS creates all the constants that are used in the composition. These can include

components, modules, bindings, integers, or strings. After this, the predicates are declared as functions using their signatures.

Following this, all the predicates themselves are added as assertions. This is where the constants and functions are used. For a predicate such as *@Firmware(thisComponent, 4, 5)* where *thisComponent* has been named #c1, *c1* would be defined as a Component in the constants declaration section the assertion. The predicate Firmware is defined as a function that takes in a component and two integers while returning a boolean value, and the result would be *(assert (Firmware c1 4 5))*. With this assertion, PCS is telling Z3 that this specific instance of Firmware is true. This continues for all predicates in the composition. Finally, PCS asserts the first order logic statements one by one, checking satisfiability each time for error reporting, using Z3.

4.4.2 Running the Z3 Code

After generating the Z3 code, the next step is to run it to see if it was satisfiable. At this point, a function takes in the translated Z3 code with the assertion. Using the Z3 Java API, a new context is created, from which the solver is created. Using a method provided by the API, the string containing the Z3 code is passed to this solver. Satisfiability is then checked using this solver, returning true for *sat*, false for *unsat*, and throwing an exception for *unknown* or, in the case where the compiler failed to catch errors, a Z3 Exception.

4.4.3 Reporting Feedback from Z3

A limitation of Z3 is being able to understand exactly what went wrong. A proof can be generated when a composition is unsat, but it does not contain any immediately useful information, as seen in Appendix C. Since PCS asserts the first order logic statements one by one, it can report back to the user which specific statement was not satisfied by the factual predicates, which at least provides some level of granularity. Feedback for the user is very important for fixing errors and unintended behaviors.

4.4.4 Ambiguity with Z3

Using Z3 extensively throughout our project, the ambiguity present in it became more apparent. The main case where it is especially prevalent is when trying to check if items exist in the composition. Z3 will try to make satisfiable solutions every time. This can be detrimental because it will make assumptions about the existence of facts.

For example, suppose we have a composition and one component requires another one to be present. The logic for this would be *EXISTS b. FORALL a. Requires(a, b).* From this logic we are stating that component *a* requires component *b* to be present in the composition. However, Z3 will not check this. Instead, Z3 will assume that *b* does exist in the composition somewhere. Due to this assumption, it is very difficult to check that the desired component is present.

More challenges arise from this assumption and logic as well. This assertion will not be checked unless an *(assert (Requires a b))* is present in the composition. In order for this assertion not to cause an error previously in Z3, *a* and *b* must both be defined.

From here, it will check the assertion and be able to satisfy it, since both exist. Essentially, the logic will not be reporting back that the component is not present. Instead, Z3 will be checking it by producing an error when the component has not been defined and not by the logic, which is undesirable.

4.5 Compiler Design

The following section outlines the high-level design of the compiler for PCS: the internal workings of what happens when it is used to compile Z3 code, and how.

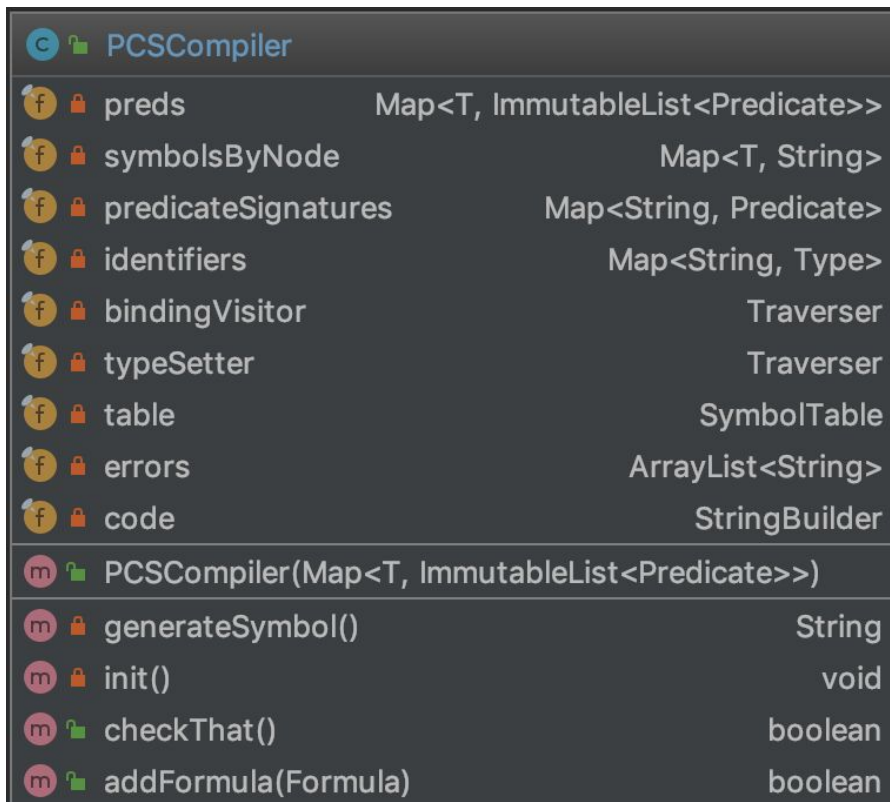


Figure 11: UML for PCSCompiler

Figure 11 shows the simple API for using PCS Compiler: the constructor, where one provides a mapping between each instance level and an ImmutableList<Predicate>

representing all the predicates attached to their respective instance level's AST Node, parsed from the comments of that node; the *addFormula* method that takes in a parsed *Formula*, and the *checkThat* method that interfaces with the Z3 API to check satisfiability or errors.

The use case for the API is as follows: a user must create an instance of the compiler with parsed predicates and their map keys, representing level instances, and check for compilation errors. If there are no errors, the compiler has produced Z3 code for the factual predicates, and the user may then proceed to add formulas. To check if these factual predicates compile to Z3 and to check satisfiability, the user must call *checkThat.*

To check formulas, a user calls *addFormula* with a parsed formula and if it returns true, then the formula has passed both compilation and Z3 satisfiability and has its respectful Z3 code added to *code*. False indicates an *unsat* status, and compiler errors are thrown otherwise.

The design of this API was intended to allow the finest grain of error reporting - all the predicates must be compiled at once to check if they are valid *together* in the domain of facts that have been asserted. Errors will surface for each compilation error found for the predicates. Additionally, the user has the power to tell which formulas, if any, have failed compilation or satisfiability. Since a program is unsat if *one* assertion is *unsat*, the best error reporting will indicate so at the individual formula granularity.

4.6 Testing

The following sections outline how the parts of the compiler were tested.

4.6.1 Lexer testing

To thoroughly test a lexer, we tested that a valid, respectful token was created for each token possible. We also tested that white-space did not get parsed. Figure 12 below shows a sample of the test cases we ran.

```
70          token = newTokenizer( s: "thisBinding").advance();
71          assertEquals(BINDING, token.type);
72          token = newTokenizer( s: "x").advance();
73          assertEquals(VAR, token.type);
74          assertEquals( expected: "x", token.text);
75          token = newTokenizer( s: "429508").advance();
76          assertEquals(NUM, token.type);
77          assertEquals( expected: "429508", token.text);
78          token = newTokenizer( s: "").advance();
79          assertEquals(EOF, token.type);
80          PCSLexer lex = newTokenizer( s: "x y");
81          assertEquals(VAR, lex.advance().type);
82          assertEquals(VAR, lex.advance().type);
83      }
84
85      @Test
86      public void testWS() {
87          Token token = newTokenizer( s: "    x").advance();
88          assertEquals(VAR, token.type);
89          token = newTokenizer( s: "\n").advance();
90          assertEquals(EOF, token.type);
91          token = newTokenizer( s: "\r").advance();
92          assertEquals(EOF, token.type);
93          token = newTokenizer( s: "\r\n").advance();
94          assertEquals(EOF, token.type);
95          token = newTokenizer( s: "\n x").advance();
96          assertEquals(VAR, token.type);
97          token = newTokenizer( s: "\n\r x").advance();
98          assertEquals(VAR, token.type);
99          token = newTokenizer( s: "\n\n x").advance();
100         assertEquals(VAR, token.type);
101     }
```

Figure 12: Lexer test cases for expected token types and whitespace ignorance

The lexer tests tokenize given input texts and assert that the type of the parsed token is the intended type. The whitespace tests on line 86 do the same operation, asserting that the whitespace does not interfere with tokenization.

4.6.2 Parser testing

Test cases for parsing is difficult, because it is not easy to automatically confirm that the structure of the ASTs are as expected. We visually confirmed the structure of the trees to be valid via a visiting printer, and also tested whether input should parse. Example test cases are shown in Figure 13 below.

```
@Test
public void testPrecedence() throws ParserException {
    testParsing( code: "P(x) => Q(y) => R(z) OR Z(a)");
}

@Test
public void testFail() {
    testParsingError( code: "x AND y");
    testParsingError( code: "x => x");
    testParsingError( code: "x => y");
    testParsingError( code: "x <=> x");
    testParsingError( code: "x <=> y");
    testParsingError( code: "~(x => y)");
    testParsingError( code: "~(~(x) => y");
    testParsingError( code: "x & x");
    testParsingError( code: "x | ~y");
    testParsingError( code: "true");
}

@Test
public void testParenthesis() throws ParserException {
    testParsing( code: "(FORALL x. x==x)");
    testParsing( code: "((FORALL x. (x==x)))");
    testParsing( code: "FORALL x. (x==x) AND (y==x)");
    testParsing( code: "(FORALL a b c. (Component(a) AND (Firmware(a,b,c))) => (b <= c))");
    testParsing( code: "(FORALL x y. ((Component(x) AND Component(y)) => " +
            "(((EXISTS a b. ((Id(x,a)) AND Id(y,b))  AND (~(a==b)))) OR x==y)))");
}

@Test
public void testQuantifiers() throws ParserException {
    testParsing( code: "FORALL x. x==x");
    testParsing( code: "FORALL x y. x!=y");
    testParsing( code: "FORALL x. EXISTS y. x!=y");
    testParsing( code: "FORALL x y. EXISTS w z. x!=y AND w!=z");
}

@Test
public void testPredicates() throws ParserException {
    testParsing( code: "Pred(a) => Pred(b) OR Pred(c) AND Pred(a)");
    testParsing( code: "FORALL a. EXISTS b c. Component(a)  => Firmware(a,b,c)");
```

Figure 13: Parsing test cases

4.6.3 Compiler Testing

The compiler was tested through factual predicates, formula, and satisfiability assertions. We tested whether the compiler threw errors when expected, or whether the Z3 compilation result was expected. We tested every type of error accounted for in the compiler. The goal for the compiler is to never allow a Z3 exception unless for the semantic failures caused by an *Unknown* return status. We are unaware of any possible exceptions not caught by PCS Compiler, however there may still be unresolved matters. Sample test cases are shown in Figures 14 and 15 below.

```
45      @Test
46      public void testValidPredicates() throws CompilerException, Z3Exception {
47          p.add("Predicate(thisModule, 5)\n");
48          p.add("Predicate2(thisModule, 6)\n");
49          p.add("Predicate3(thisModule)");
50          parseNode();
51          p.add("Predicate4(thisBinding, 5)\n");
52          p.add("Predicate5(thisBinding, 6)\n");
53          parseNode();
54          assertTrue(checkCompile());
55      }
56
57      @Test(expected = CompilerException.class)
58      public void properArity() throws CompilerException, Z3Exception {
59          p.add("Predicate()\n");
60          parseNode();
61          checkCompile();
62      }
63
64      @Test(expected = CompilerException.class)
65      public void properArityInductive() throws CompilerException, Z3Exception {
66          p.add("Predicate(thisModule)");
67          p.add("Predicate1()");
68          parseNode();
69          checkCompile();
70      }
71
72      @Test(expected = CompilerException.class)
73      public void firstArgMustBeBindingLevel() throws CompilerException, Z3Exception {
74          p.add("Predicate(5, thisModule)");
75          parseNode();
76          checkCompile();
77      }
```

Figure 14: Testing factual predicates

```
329    public void formulaMultiLevelTypeInferencingComparisonOperator() throws Z3Exception, ParserException, CompilerException {
330        p.add("thisComponent -> #c1");
331        p.add("Firmware(thisComponent, 400, 400)");
332        parseNode();
333        p.add("thisComponent -> #c2");
334        p.add("Test(thisComponent, 50)");
335        parseNode();
336        f = "FORALL a. EXISTS b c. Firmware(b, c, 400) => a < c";
337        assertTrue( condition: checkCompile() && parseFormula());
338    }
339
340    @Test(expected = CompilerException.class)
341    public void compareOnDissimilarTypes() throws Z3Exception, ParserException, CompilerException {
342        p.add("Firmware(thisComponent, 4, 5)");
343        parseNode();
344        p.add("Firmware(thisComponent, 4, 5)");
345        parseNode();
346        p.add("thisComponent -> #compName");
347        parseNode();
348        f = "FORALL a. EXISTS b c. Firmware(a,b,c) => c!=a";
349        assertTrue( condition: checkCompile() && parseFormula());
350    }
351
352    @Test
353    public void formulaEqualityInferTypeIfOneIsKnownStr() throws Z3Exception, ParserException, CompilerException {
354        p.add("Name(thisComponent, \"findsHackers\")");
355        parseNode();
356        p.add("Name(thisComponent, \"findsHackers\")");
357        parseNode();
358        f = "EXISTS y. FORALL x. Name(x,y) AND EXISTS s. s==y AND s==\"findsHackers\"";
359        assertTrue( condition: checkCompile() && parseFormula());
360    }
361
362    @Test
363    public void formulaEqualityInferTypeIfOneIsKnownNum() throws Z3Exception, ParserException, CompilerException {
364        p.add("Name(thisComponent, 5)");
365        parseNode();
366        p.add("Name(thisComponent, 5)");
367        parseNode();
368        f = "EXISTS y. FORALL x. Name(x,y) AND EXISTS s. y==s AND s==5";
369        assertTrue( condition: checkCompile() && parseFormula());
```

Figure 15: Testing Formulas

Level instance predicates are simulated via adding predicate PCS code text and registering them by calling *parseNode*. Formulas are tested by creating only one, similarly by calling *checkCompile* to run all the PCS code for compilation, where *parseFormula* then checks for Z3 compilation. The tests check every possible error we have formulated in the previous sections, as well as testing arbitrary valid formulas.

5. Conclusion

Shape Security protects its customers by employing Pegasus, a reverse proxy system that is located between the end user and the origin server. Pegasus distinguishes between malicious traffic and legitimate users through the use of policies, which are composed of many different components, and created through Policy Composer. Since the current system is not robust for policy customization, there is a need for a better way to specify how policy components interact with each other to create a valid policy composition. Currently, there are a handful of built-in annotations that component writers could include in the component being written, but no easy way to specify custom annotations.

We designed and developed Predicate-based Composer System (PCS) to address this issue by making a more flexible and customizable system for components of a policy. With PCS, predicates can be defined for components, where predicates are like facts that are true for a specific component. Using these predicates, PCS uses first order logic formulas to determine whether the logic is valid for all the predicates asserted by the components present in the policy, by assessing the composition's satisfiability using Z3.

Certain first order logic statements are applied to every component in every policy, as logic is the basis for a valid policy. Other first order logic statements can be created or customized based on the requirements from the customer for that specific policy, without needing to ask the component writers to implement the functionality to

the specific component. If the first order logic statements cannot be satisfied using Z3, PCS produces an error to inform the user which statement was unable to be satisfied. In general, PCS is a more extensive way to compose new policies through its use of first order logic and Z3 to determine if a policy composition is valid.

6. Future Work

In order for Predicate-based Policy System (PCS) to be used, it will need to be integrated with Policy Composer. The sections below details how PCS can be improved and integrated in the future.

6.1 PCS Performance Optimization

While PCS works based upon the tests that we have written, more testing to further understand the semantic translations to Z3 needs to be done; it is often difficult to translate a requirement into the logic needed to express the composer constrictions required.

Additionally, the compiler need optimizations - there are multiple iterations over ASTs that could be reduced to lower the runtime efficiency. The symbol table system needs to also be able to include separate scopes per level in an AST, instead of assuming every quantifier is either the root scope or contained in.

6.2 UI Mockups for Integration

Once our language was fully implemented, we began creating UI Mockups to visualize how PCS would interact with Policy Composer when integrated. Future work would be to use these mockups to commence integration.

The information needed in the mockup was the required predicates for every component no matter which policy they belong to, the optional predicates for when the Policy Composer users want to create additional predicates for a specific policy, and error reporting for when a predicate was violated. We created mockups for how we envision integration with Policy Composer would work, along with how errors would be reported to these mockups.

6.2.1 Mockup For Policy Composer Integration

First, the UIs would be stored in a new “Predicate Logic” tab in the top bar of the Policy Composer front end UI, as seen in Figure 9.

Figure 16: Predicate Logic Tab Mockup

Figure 10 below shows a Policy Composer UI to ensure required predicates are satisfied while being able to add or create optional ones if desired. In this UI, we present the user with all the “default predicate logic statements” that are always applied to every composition. They contain the logic and a description to go along with the logic. Directly below this is the pre-defined “optional predicate logic statements.” These are predicate logic statements that are commonly used in compositions, but do not apply to every composition. Again, the logic and description for the logic is present. These statements can be toggled “on” and “off” based on whether or not the Policy Composer user wants to apply it to their composition.

The final box (“optional predicate logic template”) is a template available for the user to write in their own statements that must be true for the composition. Similar to the optional predicate logic statements, this template contains a description, the logic, and a checkbox on whether it is applied to the composition. Clicking the “+” icon below this allows for the user to produce another template for adding more custom statements.

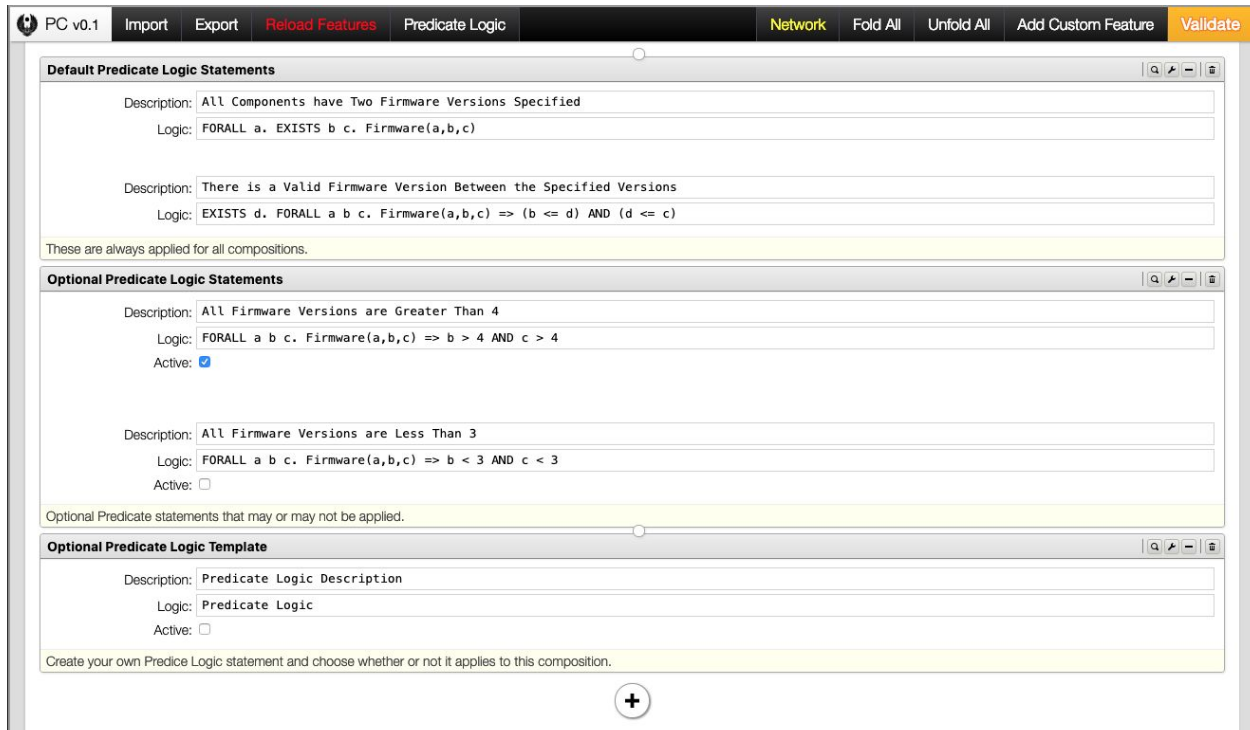


Figure 17: Predicate Logic UI Mockup

6.2.2 Mockup For Error Reporting

Reporting errors to the user is very important when introducing a new system. Due to the issues with Z3 feedback discussed in Section 4.4.3, error reporting should be as informative as possible.

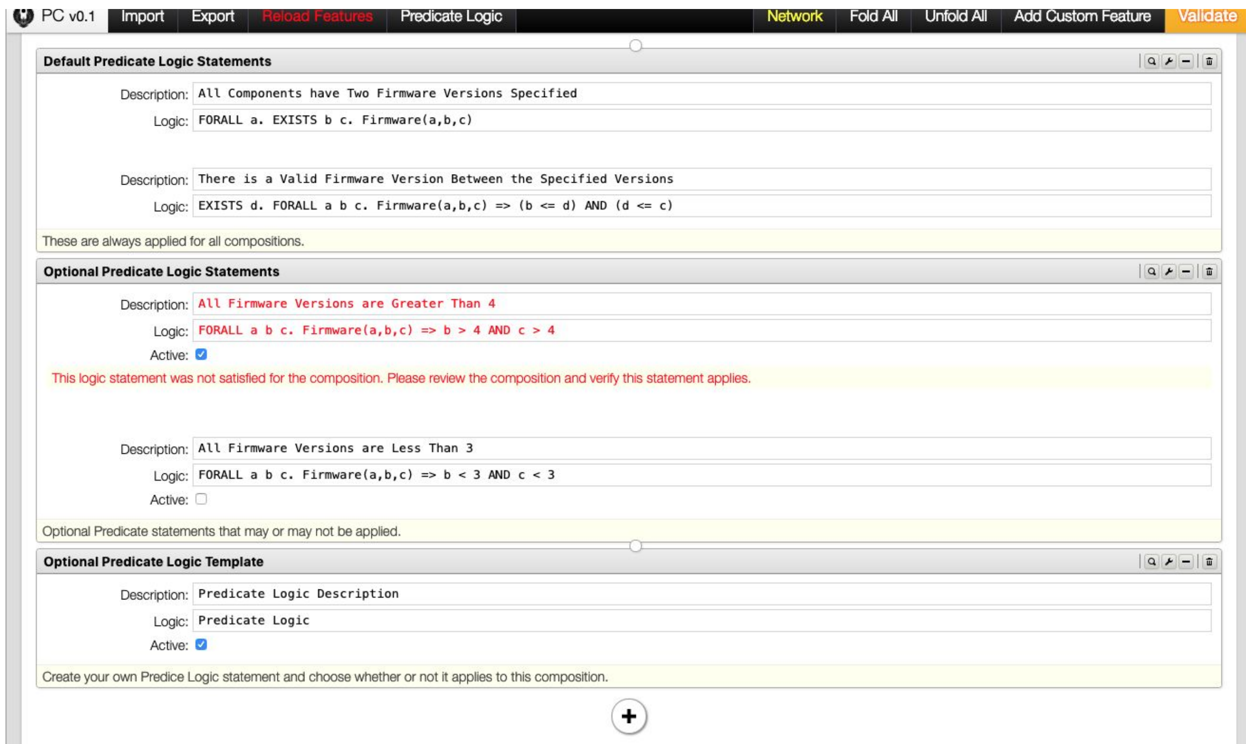


Figure 18: Predicate Logic UI Errors Mockup

Figure 11 above is a mockup of an invalid policy composition being reported to the policy composer user. The text of the predicate logic statement that was not satisfied turns red and informs the user that it was not satisfied, with some additional information based on if it is a default, optional, or created statement. The default statement error informs the user to review the composition. Building off this, the optional logic statement error asks the user to review that it is valid for the composition, since it can be toggled off. The error for the created logic statements asks the user to review the composition, the logic, and verify it applies. The varying messages are intended to remind the user what to check specifically, since there are more areas to check when adding customizability.

7. References

[1] Number of Internet Users Worldwide 2005-2017. Statista. Date Accessed: February 14, 2019, www.statista.com/statistics/273018/number-of-internet-users-worldwide/

[2] Shape Security, “Protection Against Cyberattacks and Automated Fraud”, Date Accessed: February 21, 2019, https://shapesecurity.com/

[3] Salman Saghafi, Kathi Fisler, Shriram Krishnamurthi. *Features and Object Capabilities: Reconciling Two Visions of Modularity*. ACM, 25-34, Potsdam, Germany, March 25, 2012

[4] Yakir Vizel, Georg Weissenbacher, Sharad Malik (2015). *Boolean Satisfiability Solvers and Their Applications in Model Checking*. Proceedings of the IEEE Vol. 103 Iss. 11, IEEE, 2021-2035 November 2015

[5] Nikolaj Bjorner, “Programming Z3”, Date Accessed: February 21, 2019, http://theory.stanford.edu/~nikolaj/programmingz3.html

[6] Nikolaj Bjorner, “Getting Started with Z3: A Guide”, Date Accessed: February 26, 2019, https://rise4fun.com/Z3/tutorial/guide

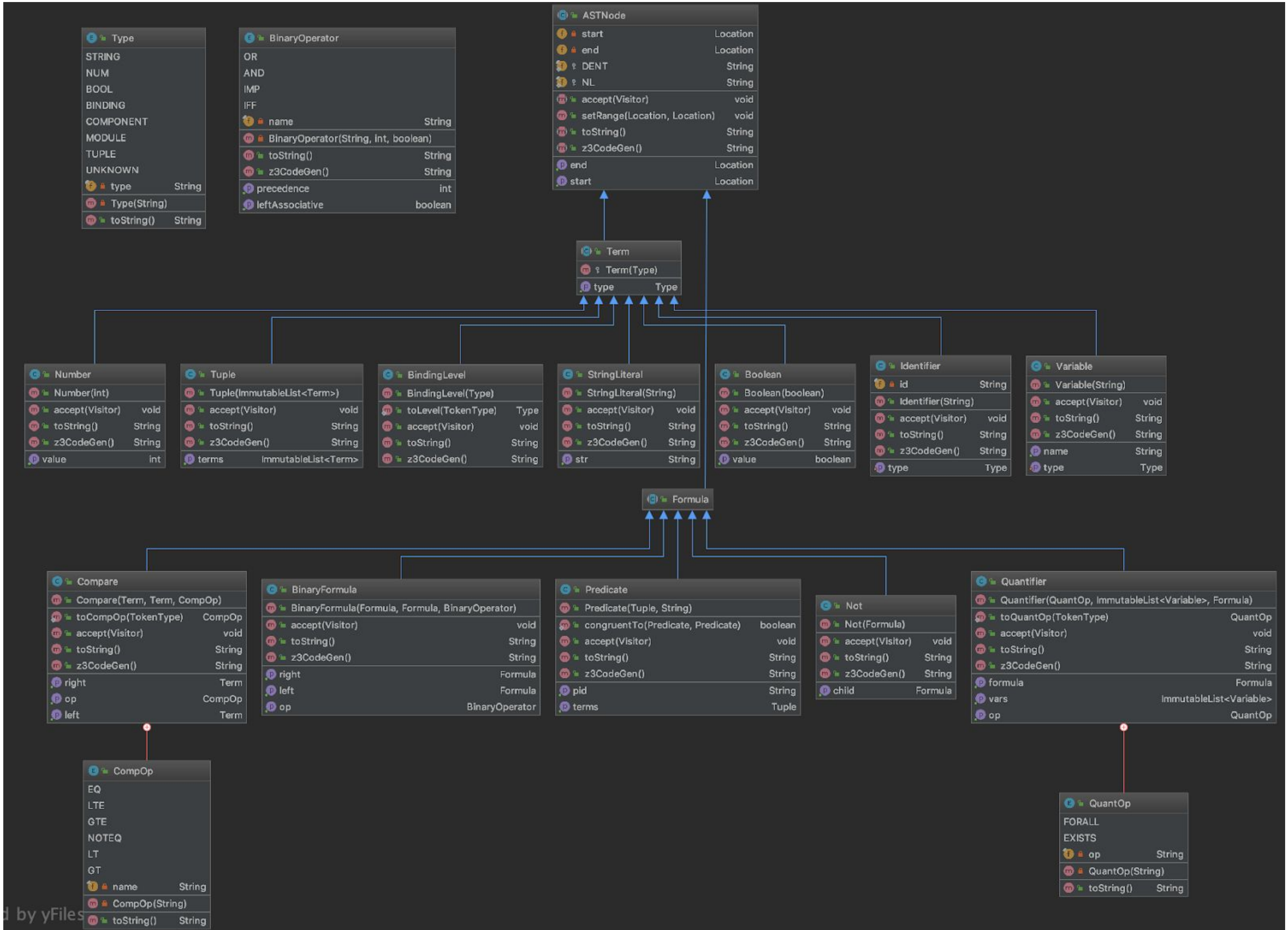
[7] Shape Security, “Functional Library for Java”, Date Accessed: February 21, 2019, https://github.com/shapesecurity/shape-functional-java

[8] Gerwin Klein, “JFlex User's Manual”, September 21, 2018, Date Accessed: February 21, 2019, http://jflex.de/manual.html

[9] Carol Wolf, “The Shunting Yard Algorithm”, Date Accessed: February 26, 2019 http://www.oxfordmathcenter.com/drupal7/node/628

Appendix A: UML Diagrams

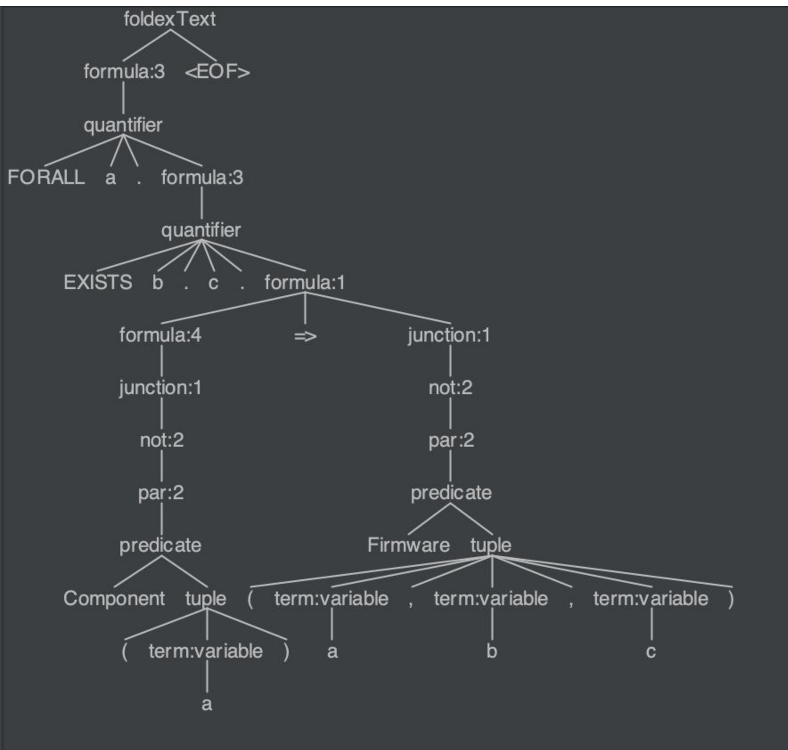
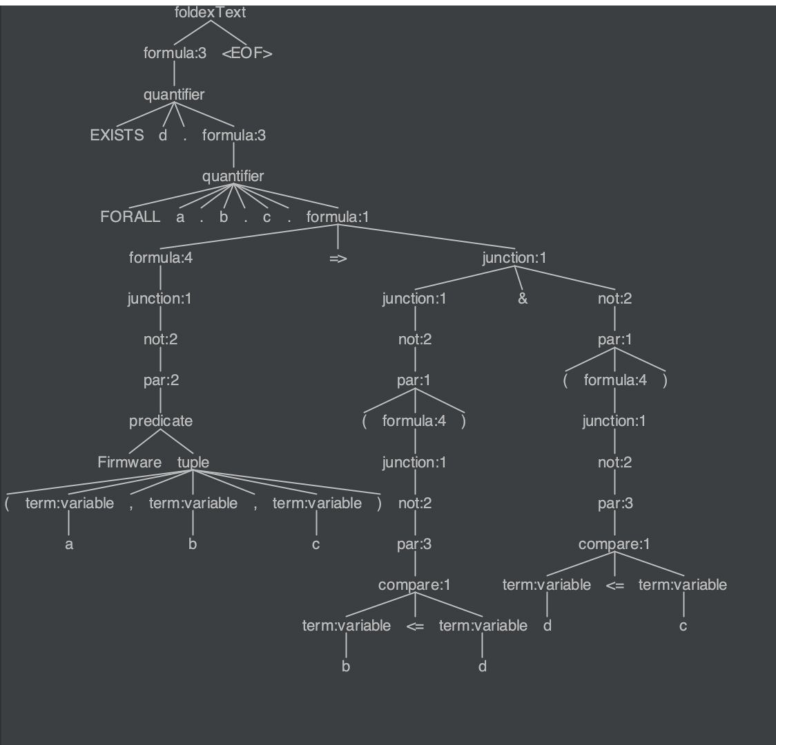
AST Node Classes

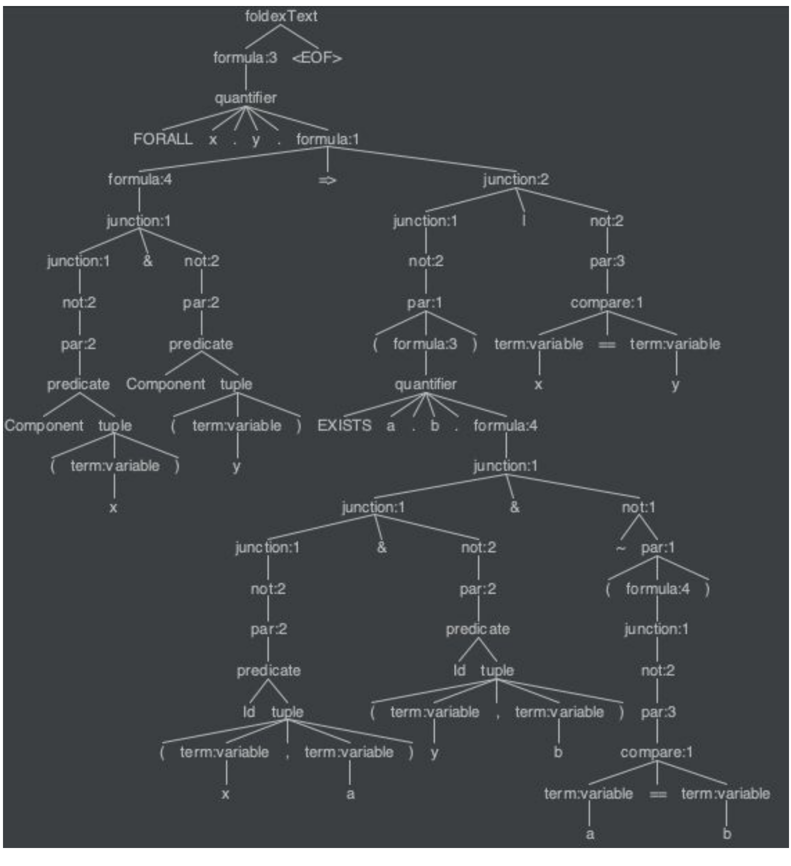
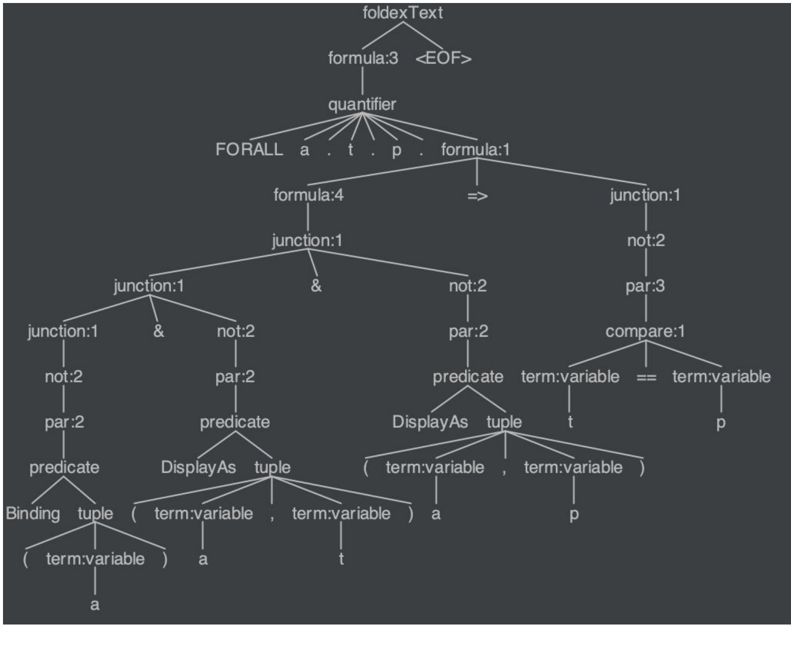


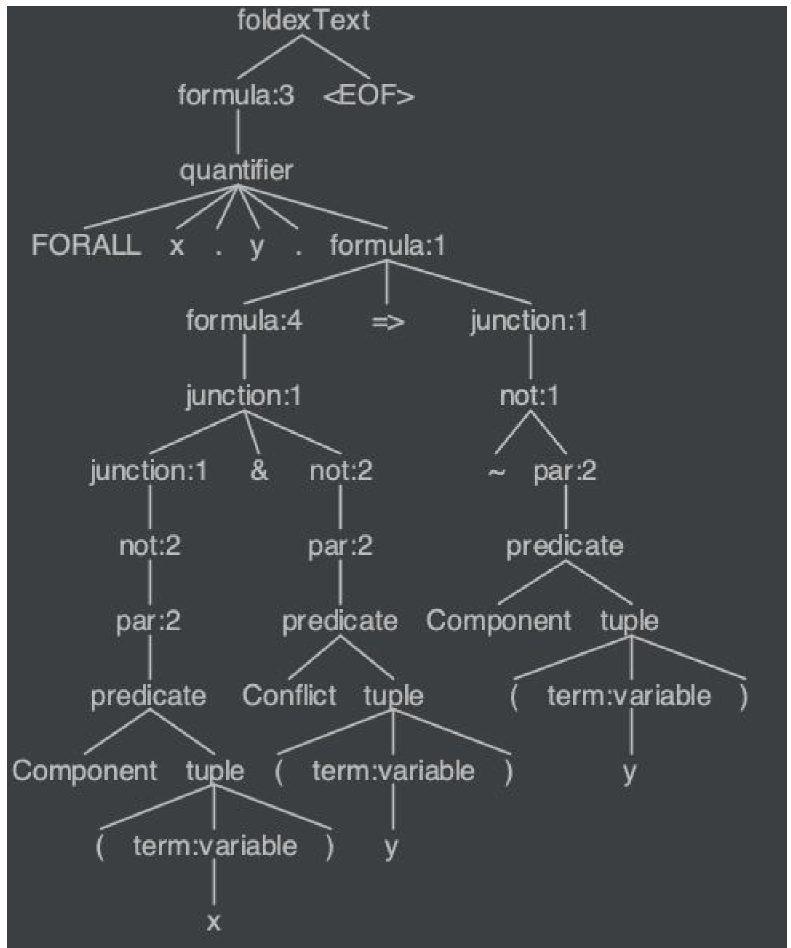
Parser Classes

Appendix B: Parsing Test Cases with ANTLR

Description	Test	Screen Cap
Valid Firmware	FORALL a b c. Component(a) & Firmware (a,b,c) => b <= c	

<table>
<tr>
<td>Firmware condition is satisfied in each component</td>
<td>FORALL a. EXISTS b c. Component(a) => Firmware(a,b,c)</td>
<td>
</td>
</tr>
<tr>
<td>There is a version between b and c</td>
<td>EXISTS d. FORALL a b c. Firmware(a,b,c) => (b <= d) & (d <= c)</td>
<td>
</td>
</tr>
</table>

<table>
<tr>
<td>Unique component only shows up once</td>
<td>FORALL x y. Component(x) & Component(y) => (EXISTS a b. Id(x,a) & Id(y,b) & ~(a==b))| x==y</td>
<td>
</td>
</tr>
<tr>
<td>Displaying a date from a binding in the UI cannot appear as a different type</td>
<td>FORALL a t p. Binding(a) & DisplayAs(a,t) & DisplayAs(a,p) => t==p</td>
<td>
</td>
</tr>
</table>

Component cannot be present if any of the components in the list are also present	FORALL x y. Component(x) & Conflict(y) => ~Component(y)	

Requires another component to be present	FORALL x y. Component(x) & Requires(y) => Component(y)	foldexText formula:3 <EOF> quantifier FORALL x . y . formula:1 formula:4 => junction:1 junction:1 not:2 junction:1 & not:2 par:2 not:2 par:2 predicate par:2 predicate Component tuple predicate Requires tuple (term:variable) Component tuple (term:variable) y (term:variable) y x

Appendix B - Table 1: Test Cases with Screen Captures

Test	*Extensive Tests (**Bold** fails)*
FORALL a b c. Component(a) & Firmware (a,b,c) => b <= c	FORALL a b c. (Component(a)) & Firmware (a,b,c) => b <= c ; FORALL a b c. (Component(a) & Firmware(a,b,c)) => b <= c ; FORALL a b c. (Component(a) & Firmware(a,b,c) => b <= c); FORALL a b c. Component(a) & Firmware(a,b,c) => (b <= c); FORALL a b c. ((Component(a) & Firmware(a,b,c)) => (b <= c)); (FORALL a b c. ((Component(a) & (Firmware(a,b,c)))) => ((b) <= c)); **((FORALL a b c. ((Component(a) & (Firmware(a,b,c))))) => ((b) <= c));**
FORALL a. EXISTS b c. Component(a) => Firmware(a,b,c)	(FORALL a. ((EXISTS b c. ((Component(a))) => (Firmware(a,b,c))))); **(FORALL a. ((EXISTS b c. ((Component(a)))) => (Firmware(a,b,c))));**
EXISTS d. FORALL a b c. Firmware(a,b,c) => (b <= d) & (d <= c)	(EXISTS d. FORALL a b c. (Firmware(a,b,c)) => ((((b)) <= d) & (d <= (c)))); **((EXISTS d. FORALL a b c. (Firmware(a,b,c))) => ((((b)) <= d) & (d <= (c))));**

<table>
<tr><td>FORALL x y. Component(x) & Component(y) => (EXISTS a b. Id(x,a) & Id(y,b) & ~(a==b))| x==y</td><td>(FORALL x y. ((Component(x) & Component(y)) => (((EXISTS a b. ((Id(x,a)) & Id(y,b)) & (~(a==b))))| x==y)));

(FORALL x y. ((Component(x) & Component(y))) => (((EXISTS a b. ((Id(x,a)) & Id(y,b)) & (~(a==b))))| x==y));

(FORALL x y. ((Component(x) & Component(y)))) => (((EXISTS a b. ((Id(x,a)) & Id(y,b)) & (~(a==b))))| x==y);</td></tr>
<tr><td>FORALL a t p. Binding(a) & DisplayAs(a,t) & DisplayAs(a,p) => t==p</td><td>FORALL a t p. Binding(a) & (DisplayAs(a,t) & DisplayAs(a,p)) => t==p;

FORALL a t p. Binding(a) & DisplayAs(a,t) & DisplayAs(a,p) => (t)==p;</td></tr>
<tr><td>FORALL x y. Component(x) & Conflict(y) => ~Component(y)</td><td>(FORALL x y. (Component(x) & Conflict(y)) => (~(Component(y))));

(FORALL x y. (Component(x) & Conflict(y))) => (~(Component(y)));</td></tr>
<tr><td>FORALL x y. Component(x) & Requires(y) => Component(y)</td><td>(((FORALL x y. (Component(x) & (Requires(y))) => Component((y)))));

(FORALL x y. (Component(x) & (Requires(y)))) => Component((y));</td></tr>
</table>

Appendix B - Table 2: List of Extensive Tests

Appendix C: Z3 Proof when Unsatisfiable

```
(proof
(let ((?x213 (cFV!29 x)))
(let ((?x422 (third ?x213)))
(let ((?x421 (second ?x213)))
(let ((?x420 (first ?x213)))
(let ((?x423 (mk-tuple ?x420 ?x421 ?x422)))
(let ((?x77 (bFV!30 x)))
(let ((?x411 (third ?x77)))
(let ((?x410 (second ?x77)))
(let ((?x251 (first ?x77)))
(let ((?x412 (mk-tuple ?x251 ?x410 ?x411)))
(let (($x216 (HasFirmware x ?x412 ?x423)))
(let ((@x578 (monotonicity (symm ((_ th-lemma datatype) (= ?x77 ?x412)) (= ?x412 ?x77)) (symm ((_ th-lemma datatype) (= ?x213 ?x423)) (= ?x423 ?x213)) (= $x216 (HasFirmware x ?x77 ?x213)))))
(let (($x214 (HasFirmware x ?x77 ?x213)))
(let (($x404 (forall ((a Feature) )(! (let (($x46 (IsFeature a)))
(let (($x146 (not $x46)))
(or $x146 (HasFirmware a (bFV!30 a) (cFV!29 a))))) :pattern ( (IsFeature a) ) :qid k!176))
))
(let (($x398 (forall ((a Feature) )(! (let (($x46 (IsFeature a)))
(let (($x146 (not $x46)))
(or $x146 (HasFirmware a (bFV!30 a) (cFV!29 a))))) :qid k!176))
))
(let (($x46 (IsFeature ?0)))
(let (($x146 (not $x46)))
(let (($x392 (or $x146 (HasFirmware ?0 (bFV!30 ?0) (cFV!29 ?0)))))
(let (($x266 (forall ((a Feature) )(! (exists ((bFV Tuple) (cFV Tuple) )(! (let (($x254 (HasFirmware a bFV cFV)))
(let (($x210 (IsFeature a)))
(let (($x211 (not $x210)))
(or $x211 $x254)))) :qid k!176))
 :qid k!176))
))
(let (($x386 (exists ((bFV Tuple) (cFV Tuple) )(! (let (($x254 (HasFirmware ?0 bFV cFV)))
(let (($x210 (IsFeature ?0)))
(let (($x211 (not $x210)))
(or $x211 $x254)))) :qid k!176))
))
(let (($x381 (forall ((a Feature) )(! (exists ((bFV Tuple) (cFV Tuple) )(! (let (($x254 (HasFirmware a bFV cFV)))
(let (($x210 (IsFeature a)))
(=> $x210 $x254))) :qid k!176))
 :qid k!176))
))
(let (($x384 (exists ((bFV Tuple) (cFV Tuple) )(! (let (($x254 (HasFirmware ?0 bFV cFV)))
(let (($x210 (IsFeature ?0)))
(=> $x210 $x254))) :qid k!176))
))
(let (($x250 (= (=> (IsFeature ?2) (HasFirmware ?2 ?1 ?0)) (or (not (IsFeature ?2)) (HasFirmware ?2 ?1 ?0)))))
(let ((@x394 (quant-intro (quant-intro (rewrite $x250) (= $x384 $x386)) (= $x381 $x266))))
(let ((@x402 (mp~ (mp (asserted $x381) @x394 $x266) (nnf-pos (sk (~ $x386 $x392)) (~ $x266 $x398)) $x398)))
(let ((@x409 (mp @x402 (quant-intro (refl (= $x392 $x392)) (= $x398 $x404)) $x404)))
(let (($x49 (IsFeature x)))
(let ((@x50 (asserted $x49)))
(let (($x76 (not $x49)))
(let (($x220 (or (not $x404) $x76 $x214)))
(let ((@x253 (mp ((_ quant-inst x) (or (not $x404) (or $x76 $x214))) (rewrite (= (or (not $x404) (or $x76 $x214)) $x220)) $x220)))
(let ((@x581 (mp (unit-resolution @x253 @x50 @x409 $x214) (symm @x578 (= $x214 $x216)) $x216)))
(let (($x217 (not $x216)))
(let (($x69 (forall ((bFV Tuple) (cFV Tuple) )(! (not (HasFirmware x bFV cFV)) :pattern ( (HasFirmware x bFV cFV) ) :qid k!81))
))
(let (($x53 (forall ((bFV Tuple) (cFV Tuple) )(! (not (HasFirmware x bFV cFV)) :qid k!81))
))
(let (($x52 (not (HasFirmware x ?1 ?0))))
(let ((@x65 (mp~ (asserted $x53) (nnf-pos (refl (~ $x52 $x52)) (~ $x53 $x53)) $x53)))
(let ((@x72 (mp @x65 (quant-intro (refl (= $x52 $x52)) (= $x53 $x69)) $x69)))
(let (($x434 (or (not $x69) $x217)))
(let ((@x435 ((_ quant-inst (mk-tuple ?x251 ?x410 ?x411) (mk-tuple ?x420 ?x421 ?x422)) $x434)))
(unit-resolution (unit-resolution @x435 @x72 $x217) @x581 false)))))))))))))))))))))))))))))))))))))))))
```