Technical Report TR08-004

Department of Computer Science
Univ. of North Carolina at Chapel Hill

A Formal Language for Training a Design Pattern Detector

Hao Xu and David Stotts

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

stotts@cs.unc.edu

March 1, 2008

A Formal Language for Training a Design Pattern Detector

Hao Xu, David Stotts
University of North Carolina
Department of Computer Science
Chapel Hill, NC 27599, USA
{xuh,stotts}@cs.unc.edu

ABSTRACT

Research in automated pattern detection from source code has focused on the efficiency of pattern extraction mechanisms; there are fewer projects on making the act of pattern definition easier and more accessible to practicing software engineers. We have developed the Program-Structured Pattern Definition Language (PsPDL) with the goal of giving programmers a familiar notation (more algorithmic, less mathematical) with which to create new pattern definitions for the catalogs that drive pattern recognition tools. The syntax of PsPDL resembles that of Java, with a few additional constructs added to express first-order or set theoretic concepts (such as quantification over identifiers); PsPDL also includes a simple module system that faciliates reusing pattern definitions. The semantics of PsPDL are based on a translation to first-order logic with a focus on modeling the dependencies among the semantic entities of OO programs. PsPDL is a first step towards solving the problem of training pattern catalogs directly in programming language source code.

1. INTRODUCTION

Pattern detection systems work from catalogs that contain pattern definitions in formal and semi-formal notations. These catalogs are not static – they need to be extended as new patterns are discovered. Writing formal pattern definitions correctly is challenging for the researchers creating the detection tools; it is certainly beyond the interest level (and often the capabilities) of the majority of practicing software engineers who would stand to benefit from the tools.

This reseach is a continuation of the SPQR project[1, 2]. In SPQR, patterns are detected in source code by "mining" the text for thousands of small, easy-to-find facts about the relationships among OO components: methods, classes, fields. For example, we might note that there is a dependency between method A and method B, since A calls B; we might find a read dependency between field X in class C and field K in class D. Patterns is SPQR are defined as the necessary and sufficient set of relationships that must exist among componets for the pattern to be said to exist. We then let a formal theorem prover determine if the facts as gleaned from the source satisfy the various pattern definitions.

Catalog Training: Patterns in SPQR are defined in a denotational semantics based on Cardelli's sigma-calculus[17]; these defintions are converted into first-order logic and combined with the facts gleaned from the source code to form input to a theorem prover. Writing SPQR pattern definitions requires mathematical sophistication and practice, and is not something many software engineers will be able to do successfully as pattern catalogs need to be extended. To assist with this problem we have developed the concept of *training the catalog.* Programmers are good at writing algorithmic code. If we can make pattern definition as much like programming as possible (and less like writing first-order formulae) then catalog extension will be a natural activity for programmers to do as new patterns are discovered.

Canonical Code: The SPQR tools take program source code and generate first-order formulae (structural and semantic facts) from it. A pattern definition is SPQR is essentially a collection of first-order formulae. Why not have programmers write small *canonical programs* that contain only the components of a pattern to be defined and let SPQR do the hard formal work? The SPQR tools can be used to glean from the canonical code the facts about its relationships; those facts would constitute the formal pattern definition. The quality of this definition would depend on how successful the programmer was in creating canonical code that contained *only* pattern components.

PsPDL: How to automatically train a pattern catalog with canonical codes in a general programming language is an open problem. In particular, an algorithm for determining what part of the SPQR-generated output is necessary (vs. extraneous) is not yet known; human post-processing is still necessary to tighten the definitions. However, we have taken a first step towards the general training solution with the research reported in this paper. We have developed the Program-Structured Pattern Definition Language (PsPDL) to provide a programming-language-like notation that is limited to expressing only pattern concepts. PsPDL gives programmers a familiar notation (more algorithmic, less mathematical) in which to create new pattern definitions for the catalogs that drive pattern recognition tools. Being a non-general notation, PsPDL definitions contain only pattern-specific information and so lead to correct complete definitions when translated into first-order language via the

SPQR tools.

2. RELATED WORK

A number of research projects have focused on how to enforce design patterns by adding new language features to generate elements that constrain a set of classes to implement design patterns. Slam-sl[3] is a specification language that can be used to specify design patterns. Based on the idea of formalizing design pattern as class operators, Slam-sl can be used as a pattern language. PaL[4] is a programming notation for design patterns that generates classes from a program language with feature that support design patterns. PEC[5] is a pattern enforcing compiler that extends a Java compiler to include the extra checks needed to enforce design patterns. These languages or language extensions helps write or generate code that matches certain design patterns, but does not directly address the problem of creating pattern definitions that can be used to discover patterns in existing source code.

Other work has been done in formalizing design patterns. LePUS3 and Class-Z[6] are a specification and modeling language designed to capture design patterns at different levels of abstraction. LePUS is a very nice language for encoding architectural descriptions, and reasoning about them. LePUS also has a visual language that show the formal definition as a graph. Abstract syntax graph are also used to define design patterns which can be should in a UML-like diagram[8]. Prolog[11] has also been used to encode design patterns and source code[7]. However, unlike general purpose theorem provers, Prolog programs are often restricted to Horn clauses. However, LePUS does not provide as much detail as PsPDL, which makes it less effective for defining patterns used to match the source code. Also, the syntax of LePUS, graph-based languages, and Prolog are closer to logical formulae than programs written in programming languages. A closely related project from SPQR is POML and EDPs [1, 2]. POML is an XML based pattern definition and source code modeling language. It encodes concepts of object-orient design and patterns in an XML dialect which can be translated to Otter theorem prover input. Finally, recent work is showing an increased effort in dynamic discovery of patterns[9].

3. THE PSPDL PATTERN LANGUAGE

The core language presented in this paper is called the Program-Structured Pattern Definition Language(PsPDL). PsPDL is designed with a programming-language-like syntax resembling that of Java (for familiarity to programmers), with a few additional constructs to allow quantification over identifiers. PsPDL also has a simple module system that facilitates reusing pattern definitions. The semantics of PsPDL are based on a translation to object calculi in First-Order Logic with Strict Variables and Indices (FOLSVI)[16]. It focuses on modeling the dependency of semantic entities rather than the actual semantics of evaluation, which is more relevant to design patterns description. It is designed as a minimal language for design patterns; a lot of features found in object-oriented programming languages are left out, and the syntax is more restrictive. However, it supports other features designed for describing design patterns, such as constructs and directives for explicitly encoding elemental design patterns (EDPs)[1]. We take a two layered approach: PsPDL has a core language that is easier to define and perform inference on; other syntax features are defined in the core to allow writing more legible definitions.

4. PSPDL SYNTAX

The syntax of PsPDL is shown in Figure 1. There are a few conventions used in the presentation which we introduce now. On the right hand side of a production, optional, multiple are represented by superscripts "?" and "*", "+", respectively. If a set of parentheses are followed by one of the multiplicity modifiers, then they are meta symbols that delimits subterms in the definition. When parentheses are not, they are used as a symbol in PsPDL, and are generally not quoted. If we define a list of terms that are separated by comma in PsPDL, we write "*,". One terminal symbol is left undefined in the grammar: "$\langle identifiers \rangle$" produces identifiers acceptable in the target language. The grammar has productions for expressions and statements, directives, patterns and classes, and first-order features.

The syntax is best illustrated with an example. Figure 2 shows the PsPDL code for two small patterns from the literature – Object Recursion and Objectifier[14, 15].

There are a few syntactic constructs that need elaboration. First, identifiers in PsPDL in a definitional position of classes, methods, and fields denote sets of identifiers in the object language. If they are followed by "+", then they denote any sets with nonzero finite number of elements, otherwise they denote singleton sets. To the contrary, other types of identifier in both the definitional and applicational positions denote identifiers in the object language. This distinction leads to defining the selection syntactic construct.

When set identifiers are used in any applicational position, they must be followed by a selection construct to select one of the elements. The syntax of selection construct looks like: n[**some** k]. Other types of identifier are not and should not be followed by a selection construct. It seems to be unnecessary and cluttered to required such explicit selection even for singleton sets. We will show how to introduce a syntax sugar to make the program more concise, but we will stick to this core language for now.

Method overriding has to be explicitly declared, though it only needs to be declared once for each method. When we declare that a set of methods overrides another set of methods, the latter are used as sets, and therefore should not be followed by a selection construct; this is also the case in the constraints in the "extends" clause of patterns.

Method invocations have default semantics which can be overridden by directives and even reversed. Using the "not" directive, however, should be done with care. For example, consider this code:

```
Handler v {
  not redirect
    inTypeFamily Abstractor[some d1]
    v.handleRequest[some k1]();
}
```

This does not match methods that do not make a recursive call. Instead, it matches code with at least one variable with no recursive call. Designing a more flexible "not" is part of the future work.

n, X	::=	identifier	*patterndef*	::=	pattern defintion
	$\langle identifier \rangle$			**pattern** $n((N^{*,}))^?$	
				$(\textbf{extends}\ n(N = set)^{*,})^*)^?\{cd^*\}$	
T, F, M, N	::=	set identifiers			
	n		*cd*	::=	class definition
	$n+$			**class** *class* $(\textbf{extends}\ t^{*,})^?\{md^*\}$	
t	::=	type	*md*	::=	member definition
	$T[\textbf{some}\ X]$			$t\ F$	field
				$\textbf{abstract}^?\ ms\ \{s\}^?$	method
m	::=	method			
	$M[\textbf{some}\ X]$		*ms*	::=	method signature
				$t\ M(pd)\ (\textbf{overrides}\ M)^?$	
f	::=	field			
	$F[\textbf{some}\ X]$		*pd*	::=	parameters defintion
				$(t\ n)^{*,}$	
s	::=	statement			
	new $t(n^{*,})$	new	*set*	::=	set
	$n.f = n;$	field assignment		N	unqualified
	$n = n.f;$	field retrival		$N.N$	qualified
	$n.m(n^{*,});$	method invocation		$set + set$	union
	$n = n.m(n^{*,});$	method retrieval		$\{set[\textbf{some}\ X]\}$	
	$n = \textbf{this}$	self			
	$n = \textbf{super}$	super object	*dir*	::=	primitive directive
	$n = n$	parameter		**delegate**	
	$t\ n\{s\}$	variable		**redirect**	
	return $n;$	return		**inObject**	
	$s_1 s_2$	sequence		**inType**	
	$\{s\}$	block		**inSuperType**	
	dir s	directive		**inTypeFamily** t	
				not	

Figure 1: Syntax of PsPDL

The scope of a local variable is explicitly delimited by curly braces. The variables that occur in the selection construct will be converted to strict first-order variables, meaning that their scope is within the whole pattern definition. They can even be exported to other pattern definitions. Therefore, we need to use a fresh variable in each new selection construct unless we intend to select an object selected before.

The module system is quite simplistic. A pattern may declare some of the identifiers that are exported and visible outside of the definition. When extending a pattern, the exported identifiers of the patterns referred may be bound to local identifiers. A pattern that extends other patterns may create mixins to classes defined in the other patterns, as shown in the sample pattern definition. The Objectifier pattern, for example, adds new constraints to the methods defined in the ObjectRecursion patterns to form two distinct classes of objects: Recursors and Terminators.

5. SEMANTICS

5.1 Predicate Generation Rules

This section defines Predicate Generation Rules (PGR) that generate the logical structural semantics of a pattern defined by PsPDL. The Rules are compositional and the object language used to define the semantics is the "Predicate Logic with Strict Variables and Indices"[16].

5.1.1 First-Order Logic with Strict Variables and Indices

In order to keep our translation rules compositional and yet allow cross references between difference formulas, we extend first-order logic with strict variable and indices. We list the syntax of our extension in Figure 3.

Our logic FOLSVI is essentially equivalent to FOL, but it solves a few problems relevant to pattern definition languages. The semantics of FOLSVI rely on converting FOLSVI to prenex normal form (PNF). Because for each FOL formula there is an equivalent FOL formula in PNF, and as we will introduce below, our additional transform rules do not affect non strict variables, our extension is a conservative extension in the sense that if FOL formulae are regarded as FOLSVI formulae, their semantics do not change.

First, *strict variables* are basically variables that are not subject to renaming. If we have a strict variable that is outside of the scope of a quantifier of the variables, it will be captured by the quantifier when in PNF. We don't allow strict variables to be quantified twice in a formula, which means that our syntactic rules do not always produce well-formed formulae, but we allow an exception that a strict variable may be existentially quantified multiple times, and the outermost quantifier will be retained while other quantifiers are discarded. The intuition is that we want strict variables to represent classes, methods, and fields that can be referenced outside of current scope. Because FOL does not have a scoping mechanism, we are essentially adding the scoping mechanism to FOL. The second production of strict variables $!n[s]$ is a qualified form of strict variable. Sometimes, we may want to encapsulate the strict variables, we add the scope construct $\Lambda\overline{s}(f)$ to FOL so that only variables

```
pattern Objectifier(
  Abstractor, ConcreteClass+,
  Client, Client.someMethod,
  Abstractor.method+, ConcreteClass+.method+)
  class Abstractor {
    abstract AT1 method+();
  }
  class ConcreteClass+ extends Abstractor {
    AT2 method+()
      overrides Abstractor.method+ {
    }
  }
  class Client {
    ConcreteClass ref;
    AT3 someMethod() {
      ConcreteClass v {
        v = ref;
        v.method[some i]();
      }
    }
  }
}
pattern ObjectRecursion(
  Handler, Recursor,
  Terminator, Initiator,
  someMethod, handleRequest)
  extends Objectifier(
    Abstractor = Handler,
    ConcreteClass+ = Recursor+ + Terminator+,
    Client = Initiator,
    Client.someMethod = Initiator.someMethod,
    Abstractor.method+ =
      Handler.handleRequest) {
    ConcreteClass+.method+ =
      Recursor+.handleRequest +
      Terminator+.handleRequest) {
  class Recursor+ {
    AT4 handleRequest() {
      Handler v {
        redirect
          inTypeFamily Abstractor[some d]
          v.handleRequest[some k]();
      }
    }
  }
  class Terminator+ {
    AT5 handleRequest() {
      not Handler v {
        redirect
          inTypeFamily Abstractor[some d1]
          v.handleRequest[some k1]();
      }
    }
  }
}
```

Figure 2: Pattern Definition of the Objectifier and Object Recursion Patterns in PsPDL

index	d	::=		index
		$\mathbf{0}$		zero
		$\mathbf{succ}(d)$		successor
strict	s	::=		strict variables
		$!n$		
		$!n[s]$		
	f	::=		formula
		$\prod e(f)$		indexed universal
		$\sum e(f)$		indexed existential
		$\Lambda\overline{s}(f)$		scope

Figure 3: Syntax of FOLSVI

$$
\begin{aligned}
!n[n_1, n_2, \ldots, n_k] &\triangleq !n[!n_1[!n_2[\ldots!n_k]]] \\
\{\forall|\prod|\forall_1|\prod_1\}e\{f\} &\triangleq \{\exists|\sum|\exists_1|\sum_1\}e(\forall n \in e(f)) \\
\{\forall|\prod\}e \in t(f) &\triangleq \{\forall|\prod\}e(e \in t \rightarrow f) \\
\{\exists|\sum\}e \in t(f) &\triangleq \{\exists|\sum\}e(e \in t \wedge f) \\
\{\exists_1|\sum_1\}e(f) \in &\triangleq \{\exists|\sum\}e(Singular(e) \wedge f)) \\
\{\exists_+|\sum_+\}e(f) \in &\triangleq \{\exists|\sum\}e(Plural(e) \wedge f)) \\
Singular(e) &\triangleq \exists s_e(e = \{s_e\}) \\
Plural(e) &\triangleq \exists s_e(s_e \in e)
\end{aligned}
$$

Figure 4: Abbreviation of FOLSVI

$\overline{s}$ are visible outside the construct; other strict variables are converted to variables beyond this scope. All non-strict free variables are existentially quantified.

The indices are easier to understand. The indices are like de Bruijn indices, exception that they only index variables in the indexed universal and existential. This construct is usefully for defining rules that are compositional.

The translation of FOLSVI to FOL is a two step translation. The first step converts all indices to the variables indexed. The second step converts FOLSVI to PNF. If a scope construct with empty list of strict variables is implicitly added to each FOLSVI formula, then each well-formed FOLSVI formulate can be translated into a PNF of FOL, and each FOL formula can also be regarded as a well-formed formula of FOLSVI. The formal definition of the translation rules are defined in [16].

For this paper we introduce a few abbreviations. To disambiguate, when we write in the abbreviated forms we omit the strictness modifier "!" in the brackets, for example, $!n[n_1] = !n[!n_1]$.

5.1.2 Meta-functions

Keyword in Sans Serif are meta-predicates that are used at translation time. There is also a typing environment E of local variables. All local variables have to be declare by variable definition statements. E is a map that maps local variables and to their declared type. The semantics of meta-predicates are summarized informally as follows:

- N singular: N is in the singular form.
- N plural: N is in the plural form.
- n param: n is a parameter.
- n var: n is a local variable.
- n set: n is a set identifier.
- $F[\overline{N'}/\overline{N}]$: Syntactically substitute strict variables $\overline{N'}$ for $\overline{N}$ in F.

5.1.3 Preprocessing

The preprocessing step are:

- Transforming a program to canonical form; this step de-sugars the pattern definition (which is not necessary if the pattern is written in the core language).

$$\frac{N \text{ set}}{\|N\|_s = !N}(\text{Set})$$

$$\frac{N_1 \text{ set}}{\|N_2.N_1\|_s = !N_1[N_2]}(\text{QualifiedSet})$$

$$\frac{X \text{ occurs in some class definition}}{\|\{N[\textbf{some } X]\|_s = singleton(!X)}(\text{Singleton})$$

$$\frac{\|s_1\|_s = N_1 \quad \|s_2\|_s = N_2}{\|s_1 + s_2\|_s = union(N_1, N_2)}(\text{Union})$$

Figure 5: Semantics of Set Terms

- Gathering enough information to evaluate the meta-predicates such as n param and n var; these predicates can be decided from the syntactic analysis.
- Checking that all identifiers are properly defined in scope; this can also be checked syntactically.
- Checking that no local variables are assigned twice.
- Checking that the specification is well-typed. This step should be done to ensure that a specification can match some program; however, it is not essential as there is no problem in progress as in normal programming languages since we are defining the semantics for a pattern.

5.1.4 Judgments

E	$\vdash$	$\|A\| \Rightarrow P$	the logical structural semantics of A is P
E	$\vdash$	$n : T$	the local varible n has type T
		$\|A\|_s = T$	the semantics of set expression A is T

5.1.5 Rules

First we define the semantics of set terms: $\|\circ\|_s$.

The Set rule basically translates a set identifiers to a strict variable, which are variables that are not subject to renaming. We use the same name for the variable in the object language (refering FOLSVI from here on). The Qualifed-Set rules do similarly, but construct a strict variable using the qualifier. The Singleton rule selects a element X from the set N. The reason for the requirement that X occurs in some class definitions is that we can not generate a logical formula here, because the semantics of set are terms. The identifier before the selection construct is not reflected in the semantics, but merely used as an annotation that X should come from this set. In the current semantics, there is no rules for ensuring that the specification is well-typed because progress is not a problems as we mention before. However, there is good reasons that we enforce certain type system that ensure that the patterns are not absurd and that catches unintended errors. We will discuss this point in the future work section.

The rules for meta-predicates are generally easy. For example:

$$\frac{}{n \text{ singular}}(\text{Singular})$$

$$\frac{}{n + \text{ plural}}(\text{plurar})$$

Now, we define the PGRs of PsPDL: $\|\circ\|$. The rules are composed of three sections dealing with different aspects of the semantics of PsPDL, respectively. The Structural Section, which consists Rules Specification to Rule Local Variable, addresses the structures of classes, objects, methods, and fields. Specifically, there are two rules for classes, methods, and field. The singular version defines a pattern of one class, while the non-singular version defines a pattern for a set of classes. The Dependency Section, consisting of Rules Self to Negation, addresses the dependency relations. The Similarity Section, consisting of the rest of the rules, addresses the similarity constraints.

We highlight a few rules here. Note that quantifiers do not occur in antecedents of implications, which means that we do not need to flip quantifiers when converting to PNF, unless there is a "not" directive attached to the statements; this does not affect classes, methods, or objects. We again assume that all formulae are converted to PNF.

The Specification and Pattern rules generate formulae in the form of an implication $P \rightarrow M$. If we want to query if some pattern matches some code, we can generate a logical formula from the code A, and the pattern M, and ask a theorem prover to prove that $A \rightarrow M$ is a tautology. We can see that if we can prove that the query is a tautology, then we show that the code has the pattern. The scope construction converts all strict variables that are not exported to normal variables and rewrites all occurrences of the converted strict variables to corresponding normal variables. A pattern is essentially the scope for all strict variables that are not exported.

The Extension rule generates constraints in the form of conjunction. These constraints connect two formulae generated from different pattern definitions together. To see why that works, consider formula generated from a pattern that extends another pattern. The formula looks like

$$(P \rightarrow (M \wedge (\overline{S'} \wedge \ \ M'' \ )) \wedge P' \rightarrow M'$$

The rewrite function rewrites term M' and equality constraints S by renaming the variables to get a customized copy of M'. A logical consequence of the formula is

$$P \rightarrow (M \wedge (\overline{S'} \wedge \ \ M'' \ ))$$

which is exactly what we would like to obtain from linking the two patterns together. If all variables occurred in S are strict then this rewriting step does not affect the normal variables, hence does not violate the scoping rules. Note that if without renaming the variable, when there are two extensions to the same pattern, the formula leads to a pattern that is practically too restrictive and sometime does not match any program. If instead rewriting S into M', then the resulting formula could be malformed as the right hand side may be non variables.

The Class rules defines patterns that match classes. Recall that the $\prod$ symbol is for variables that are universally quantified and indexed, and that the indices in Sans Serif refer to these indexed variables. The curly braces denote that the set is existentially quantified. For example, $\prod !class\{Class(0)\}$ is equivalent to the following $\exists !class \forall C \in class(Class(C))$. The strict variables followed by brackets $!O[C]$ are qualified strict variables. The term in the brackets must be strict variables. We also omit the exclamation symbol in the brackets. The *object* function denotes the set of objects of a class. Once the class rules is understood the other structural rules are quite similar.

The LocalVariable rules use a type to declare a local variable and add it to the typing context. The variable $!X$ is

$$\frac{\vdash \|\overline{P}\| \Rightarrow \overline{M}}{\vdash \|\textbf{specification}\ n\ \overline{P}\| \Rightarrow Specification(spec(n)) \rightarrow \bigwedge \overline{M}}(\text{Specification})$$

$$\frac{\vdash \left\|\textbf{extends}\ \overline{n(\overline{N_1} = \overline{set})}\right\| \Rightarrow I \quad \vdash ||\overline{cd}|| \Rightarrow \overline{M}}{\vdash \left\|\textbf{pattern}\ n(\overline{N})\ \textbf{extends}\ \overline{n(\overline{N_1} = \overline{set})\{\overline{cd}\}}\right\| \Rightarrow \Lambda\overline{N}(\ \ Pattern(n) \rightarrow \bigwedge \overline{M}\ \ )}(\text{Pattern})$$

$$\frac{\|set\|_s = S}{\vdash \|N = set\| \Rightarrow \ \ N = S}(\text{Constraint})$$

$$\frac{\vdash \left\|\overline{\overline{N} = \overline{set}}\right\| \Rightarrow \overline{S} \quad \vdash \left\|\textbf{pattern}\ n(\overline{N})\ pd\right\| \Rightarrow \Lambda\overline{N}(\ \ Pattern(n) \rightarrow \bigwedge \overline{M}\ \ ) \quad \overline{N'}\ \textsf{fresh}}{\vdash \left\|\textbf{extends}\ \overline{n(\overline{N} = \overline{set})}\right\| = \Lambda(\overline{S} \wedge \ \ \bigwedge \overline{M}\ \ )[\overline{N'}/\overline{N}]}(\text{Extention})$$

$$\frac{E \vdash \left\|\textbf{extends}\ \overline{class}\right\| \Rightarrow I \quad E \vdash ||\overline{md}|| \Rightarrow \overline{M} \quad class\ \textsf{plural}}{E \vdash \left\|\textbf{class}\ class\ \textbf{extends}\ \overline{class}\{\overline{md}\}\right\| \Rightarrow \begin{array}{l}\sum_{+}!class\{I \wedge Class(0)\\ \wedge \prod !O[0] \in object(0)\\ \{Instance(1,0) \bigwedge \overline{M}\}\}\end{array}}(\text{Class})$$

$$\frac{E \vdash \left\|\textbf{extends}\ \overline{class}\right\| \Rightarrow I \quad E \vdash ||\overline{md}|| \Rightarrow \overline{M} \quad class\ \textsf{singular}}{E \vdash \left\|\textbf{class}\ class\ \textbf{extends}\ \overline{class}\{\overline{md}\}\right\| \Rightarrow \begin{array}{l}\sum_{1}!class\{I \wedge Class(0)\\ \wedge \prod !O[0] \in object(0)\\ (Instance(1,0) \bigwedge \overline{M})\}\end{array}}(\text{Class Singular})$$

$$\frac{}{E \vdash \left\|\textbf{extends}\ \overline{T[\textbf{some}\ X]}\right\| \Rightarrow \exists\overline{X} \in \overline{T}(Inherits(0, !\overline{X}))}(\text{Inheritance})$$

$$\frac{E \vdash ||mbody|| = M \quad E \vdash ||pd|| \Rightarrow P \quad n\ \textsf{plural} \quad E \vdash \|o\| = N}{E \vdash \|T[\textbf{some}\ X]\ n(pd)\ o\ mbody\| \Rightarrow \begin{array}{l}\sum_{+}!n[1]\{Method(2)\\ \wedge Member(2,1)\\ \wedge \exists !X \in !T(ReturnType(2, !X)\\ \wedge N \wedge P \wedge M\}\end{array}}(\text{Method})$$

$$\frac{E \vdash ||mbody|| \Rightarrow M \quad E \vdash ||pd|| \Rightarrow P \quad E \vdash \|o\| \Rightarrow N \quad n\ \textsf{singular}}{E \vdash \|T[\textbf{some}\ X]\ n(pd)\ o\ mbody\| \Rightarrow \begin{array}{l}\sum_{1}!n[1]\{Method(2)\\ \wedge Member(2,1)\\ \wedge \exists !X \in !T(ReturnType(2, !X)\\ \wedge N \wedge P \wedge M\}\end{array}}(\text{Method Singular})$$

$$\frac{E \vdash ||pd|| = P \quad E \vdash \|o\| = N \quad n\ \textsf{plural}}{E \vdash \|\textbf{abstract}\ T[\textbf{some}\ X]\ n(pd)\ o\| = \begin{array}{l}\sum_{+}!n[1]\{Method(2)\\ \wedge Member(2,1)\\ \wedge \exists !X \in !T(ReturnType(2, !X))\\ \wedge N \wedge P \wedge Abstract(2)\}\end{array}}(\text{Abstract Method})$$

$$\frac{E \vdash ||pd|| \Rightarrow P \quad E \vdash \|o\| \Rightarrow N \quad n\ \textsf{singluar}}{E \vdash \|\textbf{abstract}\ T[\textbf{some}\ X]\ n(pd)\ o\| \Rightarrow \begin{array}{l}\sum_{1}!n[1]\{Method(2)\\ \wedge Member(2,1)\\ \wedge \exists !X \in !T(ReturnType(2, !X))\\ \wedge N \wedge P \wedge Abstract(2)\}\end{array}}(\text{Abstract Method Singular})$$

$$\frac{}{E \vdash \left\|\overline{T}[\textbf{some}\ \overline{X}]\ \overline{n}\right\| \Rightarrow \begin{array}{l}\exists !\overline{n}[2](MethodParameter(2, !\overline{n}[2])\\ \wedge \exists !\overline{X} \in !\overline{T}(Type(!\overline{n}[2], !\overline{X}))\end{array}}(\text{Method Parameter})$$

$$\frac{\|m\|_s = N}{E \vdash \|\textbf{overrides}\ m\| \Rightarrow \ \ \exists X \in N(Overrides(2, X))}(\text{Overriding})$$

$$\frac{n\ \textsf{plural}}{E \vdash \|T[\textbf{some}\ X]\ n\| \Rightarrow \begin{array}{l}\sum !n[1]\{Field(2)\\ \wedge Member(2,1)\\ \wedge \exists !X(!X \in !T \wedge ReturnType(2, !T)\}\end{array}}(\text{Field})$$

$$\frac{n\ \textsf{singular}}{E \vdash \|T[\textbf{some}\ X]\ n\| \Rightarrow \begin{array}{l}\sum !n[1]\{Field(2)\\ \wedge Member(2,1)\\ \wedge \exists !X(!X \in !T \wedge ReturnType(2, !T)\}\end{array}}(\text{Field Singular})$$

Figure 6: Structural Section of PsPDL Semantics

$$\frac{E, n :!X \vdash ||s|| \Rightarrow N}{E \vdash \|T[\textbf{some } X]\ n\{s\}\| \Rightarrow \exists n(\exists !X \in !T(Type(n, !X) \wedge N))}\text{(Local Variable)}$$

$$\frac{n \text{ var}}{E \vdash ||n = \textbf{this}|| \Rightarrow n = 1 \wedge 2 <_{use} n}\text{(Self)}$$

$$\frac{n \text{ var} \quad E \vdash n_1 : T_1 \quad Precede(T, T_1)}{E \vdash ||n = \textbf{super}(T[\textbf{some } X])|| \Rightarrow \exists !X \in !T(n \in object(X)) \wedge 2 <_{use} n}\text{(Superobject)}$$

$$\frac{n_1 \text{ var} \quad n_2 \text{ param}}{E \vdash ||n_1 = n_2|| \Rightarrow \begin{array}{l} n_1 <_{updateByParameter} !n_2[2]|_2 \\ \wedge 2 <_{use} n_1, !n_2[2]) \end{array}}\text{(Parameter Assignment)}$$

$$\frac{n_1, n_2 \text{ var} \quad E \vdash n_1 : T}{E \vdash ||n_1.F[\textbf{some } X] = n_2|| \Rightarrow \begin{array}{l} !O[T] = n_1 \rightarrow \\ \exists !X \in !F[O, T](\\ !X <_{updateByVariable} n_2|_2 \\ \wedge 2 <_{use} !X, n_2) \end{array}}\text{(Field Assignment)}$$

$$\frac{n_1, n_2 \text{ var} \quad E \vdash n_1 : T}{E \vdash ||n_2 = n_1.F[\textbf{some } X]|| \Rightarrow \begin{array}{l} !O[T] = n_1 \rightarrow \\ \exists !X \in !F[O, T](\\ n_2 <_{update} !X|_2 \\ \wedge 2 <_{use} !X, n_2) \end{array}}\text{(Field Retrieval)}$$

$$\frac{n_1, \overline{n} \text{ var} \quad E \vdash n_1 : T}{E \vdash ||n_1.M[\textbf{some } X](\overline{P} = \overline{n})|| \Rightarrow \begin{array}{l} !O[T] = n_1 \rightarrow \\ \exists !X \in !M[O, T](\\ 2 <_{invoke} !X \\ \wedge \overline{!P[M, O, T]} <_{updateByVariable} \overline{n}|_{!M[O[T]]} \\ \wedge 2 <_{use} \overline{n}) \end{array}}\text{(Method Invocation)}$$

$$\frac{n_1, n_2, \overline{n} \text{ var} \quad E \vdash n_1 : T}{E \vdash ||n_2 = n_1.M[\textbf{some } X](\overline{P} = \overline{n})|| \Rightarrow \begin{array}{l} !O[T] = n_1 \rightarrow \\ \exists !X \in !M[O, T](\\ n_2 <_{updateByMethod} !X|_2 \\ \wedge 2 <_{invoke} !X \\ \wedge \overline{!P[M, O, T]} <_{updateByVariable} \overline{n}|_{!M[O[T]]} \\ \wedge 2 <_{use} n_2, \overline{n}) \end{array}}\text{(Method Retrieval)}$$

$$\frac{n \text{ var}}{E \vdash ||n_1 = \textbf{new } T(\overline{P} = \overline{n})|| \Rightarrow \begin{array}{l} n_1 \in object(!T) \\ \wedge n_1 <_{updateByCreation} !T|_2 \\ \wedge 2 <_{create} !T \\ \wedge \overline{!P[T, O, T]} <_{updateByVariable} \overline{n}|_{!T[O[T]]} \\ \wedge 2 <_{use} n_1, \overline{n} \end{array}}\text{(Object Creation)}$$

$$\frac{n \text{ var}}{E \vdash ||\textbf{return } n_1|| \Rightarrow 2 \rightsquigarrow n_1 \wedge 2 <_{use} n_1}\text{(Return)}$$

Figure 7: Expression Section of PsPDL Semantics

$$\frac{E \vdash ||s|| = N}{E \vdash \|\textbf{not } s\| \Rightarrow \neg N}\text{(Negation)}$$

$$\frac{E \vdash ||dir^*\ n_1.M[\textbf{some } X](\overline{n})|| \Rightarrow N \quad E \vdash n_1 : T}{E \vdash \|\textbf{delegate } dir^*\ n_1.M[\textbf{some } X](\overline{n})\| \Rightarrow N \wedge \exists !X(!X \in !M[O, T] \wedge D(2, X))}\text{(Delegate)}$$

$$\frac{E \vdash ||dir^*\ n_1.M[\textbf{some } X](\overline{n})|| \Rightarrow N \quad E \vdash n_1 : T}{E \vdash \|\textbf{redirect } dir^*\ n_1.M[\textbf{some } X](\overline{n})\| \Rightarrow N \wedge \exists !X(!X \in !M[O, T] \wedge S(2, X))}\text{(Redirect)}$$

$$\frac{E \vdash ||n_1.M[\textbf{some } X](\overline{n})|| \Rightarrow N \quad E \vdash n_1 : T}{E \vdash \|\textbf{inObject } n_1.M[\textbf{some } X](\overline{n})\| \Rightarrow N \wedge n_1 = 1 \wedge IO(1, n_1)}\text{(InObject)}$$

$$\frac{E \vdash ||n_1.M[\textbf{some } X](\overline{n})|| \Rightarrow N \quad E \vdash n_1 : T}{E \vdash \|\textbf{inType } n_1.M[\textbf{some } X](\overline{n})\| \Rightarrow N \wedge IT(1, n_1)}\text{(InType)}$$

$$\frac{E \vdash ||n_1.M[\textbf{some } X](\overline{n})|| \Rightarrow N \quad E \vdash n_1 : T}{E \vdash \|\textbf{inSuperType } n_1.M[\textbf{some } X](\overline{n})\| \Rightarrow N \wedge IS(1, n_1)}\text{(Extend)}$$

$$\frac{E \vdash ||n_1.M[\textbf{some } X](\overline{n})|| \Rightarrow N \quad E \vdash n_1 : T}{E \vdash \quad \textbf{inTypeFamily}(T_1[\textbf{some } X_1])\ n_1.M[\textbf{some } X](\overline{n}) \ \Rightarrow N \wedge \exists !X_1 \in !T_1(IF(1, n_1, !X_1))}\text{(InTypeFamily)}$$

Figure 8: Directive Section of PsPDL Semantics

$$\frac{}{IO(n,n)}\text{(InObjectAxiom)}$$

$$\frac{}{n_1 \in object(T) \wedge n_2 \in object(T) \rightarrow IT(n_1,n_2)}\text{(InTypeAxiom)}$$

$$\frac{}{n_1 \in object(T_1) \wedge n_2 \in object(T_2) \wedge Precede(T_1,T_2) \rightarrow IS(n_1,n_2)}\text{(InSuperTypeAxiom)}$$

$$\frac{}{n_1 \in object(T_1) \wedge n_2 \in object(T_2) \wedge Precede(T,T_1) \wedge Precede(T,T_2) \rightarrow IF(n_1,n_2,T)}\text{(InTypeFamilyAxiom)}$$

$$\frac{m,n \text{ similar}}{S(m,n)}\text{(SimilarityAxiom1)}$$

$$\frac{\neg(m,n \text{ similar})}{D(m,n)}\text{(SimilarityAxiom1)}$$

$$\frac{}{n \in object(T_2) \wedge Precede(T_1,T_2) \rightarrow Type(n,T_1)}\text{(TypeAxiom)}$$

$$\frac{}{Precede(C,C)}\text{(PrecedeAxiom1)}$$

$$\frac{}{Inherits(C,D) \rightarrow Precede(C,D)}\text{(PrecedeAxiom2)}$$

$$\frac{}{Precede(C,D) \wedge Precede(D,E) \rightarrow Precede(C,E)}\text{(PrecedeAxiom3)}$$

Figure 9: Definitions of Auxiliary Predicates

strict. Therefore, it is not subject to renaming, which allows us to quantify the variable as eagerly as possible. If the strict variable $!X$ occurs elsewhere, only the more general quantifier is retained when converting to the prenex normal form according to the rules of FOLSVI.

The rules of assignments basically generate dependency predicates and equalities between local variables and objects. The equality constraint implies that local variables are allowed to be assigned only to one identical objects, which corresponds to local bindings in some functional programming language. However, we can simulate multiple assignment by renaming the variable at each new assignment. The rules for directives generate the predicates that overrides the default interpretation of the dependency predicates. We also need the auxiliary axioms for completeness of the definition. The similarity predicate is left undefined. They corresponds to the similarity principles of SPQR. As explained in [1], there are many ways to define similarities in methods, so we postpone the definition of similarity of methods to the application.

5.2 Predicates and Functions

Table 1 shows the predicates used in the semantics and explain what they mean. The similarity constraints are based on the similarity principles and EPDs, with a few refinements for the purpose of generality. Table 2 lists predicates used for general OOP structures and their semantics.

Some predicates and functions may seem to be redundant and can be defined in terms of the predicates listed and set theoretic predicates. For example, we may choose to define

$$object(T) \triangleq \text{collection of } M \text{ s.t. } Type(M,T)$$

The semantics will have to change a little bit. First the collection of objects is no longer necessarily a set. More importantly, all objects of the same type have to have the same methods. If one method overrides another, then this would lead to a double constraint which may not match any actual code. In the object as set definition, however, an object of some type does not necessarily belong to the object set of that type. In order to cast an object in the object set of some type to an object in the object set of the super type, we can use the assignment of super object construct. Also, when we define a local variable as referring to objects of some type, we do not assume that the object is a member of the object sets of that type, which corresponds to dynamic binding. This effectively avoided the problems incurred by defining the collection of objects of some type as equivalent to the collection of objects that has that type.

Others do not make critical difference semantically and are essentially redundant, which we use for clarity. For example, we can define

$$Instance(M,T) \triangleq Type(M,T)$$

to make it closer to the "instanceof" keyword of Java. Al-

Predicates	What the predicates denote
$S(U,V)$	Similar Member
$D(U,V)$	Dissimilar Member
$IO(U,V)$	Same object
$IT(U,V)$	Same type
$IS(U,V)$	Subtype
$IF(U,V,T)$	In the same type family
$U <_{invoke} V$	Invocation dependency
$U <_{use} V$	Use dependency
$U <_{updateByParameter} V\|_S$	Update by Parameter dependency in method S
$U <_{updateByVariable} V\|_S$	Update by Variable dependency in method S
$U <_{updateByField} V\|_S$	Update by Field dependency in method S
$U <_{updateByMethod} V\|_S$	Update by Method dependency in method S
$U <_{updateByCreation} V\|_S$	Update by Creation dependency in method S
$U <_{create} V$	Creation dependency

Table 1: Dependency Predicates

Predicates	What the predicates denote
$Method(M)$	M is method.
$Parameter(f)$	f is parameter.
$Class(C)$	C is a class.
$Abstract(M)$	M is abstract.
$Field(M)$	M is a field.
$Member(O, M)$	M is a method of O.
$ReturnType(M, T)$	The return type of M is T.
$MethodParameters(M, f)$	f is a parameter of M.
$Inherits(C, D)$	C is a subclass of D.
$Instance(O, C)$	O is an instance of C.
$Type(M, T)$	The type of M is T.
$Overrides(M, N)$	M overrides N.

Table 2: OOP Structural Predicates

though the semantics change a little bit, there is not much impact on the correctness of how the predicate are to be used.

5.3 Simplifying Singleton Constraints

As some patterns contain identifiers that denote singleton sets, the semantics for those identifier can be simplified from the generated logical formula. We use a technique called paramodulation, and discard constraints that involve variables that are not used, we can proof the following theorem.

Suppose that $F[\circ]$ is a closed FOL formula with a with a hole in a covariant position, G is an FOL formula, v is free in F, and t is a term in which no variable is captured in G. We have

$$\exists v F[v = t \wedge G] \leftrightarrow F[G[t/v]]$$

5.4 Examples

We will present a few example of simple pattern definitions and translate them to formulae of FOLSVI. The source code are translated in a similar style of the PsPDL. The translation of code considers identifers to be constants instead of variables. In the following examples, we will use some of the extended syntax introduced in the discussion.

5.4.1 Example 1: Class

We first start with a class with a single method.

```
class Abstractor+ {
  abstract AT1 method+();
}
```

The translated formula looks like

$$\begin{array}{l}\sum\nolimits_{+}!Abstractor\{Class(0)\\ \quad\wedge\prod!O[0] \in object(0)\{Instance(1,0)\\ \quad\wedge\sum\nolimits_{+}!method[1]\{Method(2)\wedge Member(2,1)\\ \quad\wedge\exists!X \in !AT1(ReturnType(2,!X)\wedge Abstract(2)\}\}\end{array}$$

We convert it to FOL formula and make it easier to read by renaming the variables.

$$\begin{array}{ll}A = & \exists AT1\exists Abstractor(\forall C(\forall O(\exists method(\forall M(\exists X(\\ & \quad Plural(Abstractor)\wedge C \in Abstractor \rightarrow\\ & \quad\quad (Class(C)\wedge(O \in object(C)))\rightarrow\\ & \quad\quad (Instance(O,C)\wedge Plural(method)\\ & \quad\quad \wedge M \in method \rightarrow\\ & \quad\quad \wedge Method(M)\wedge Member(M,O)\wedge X \in AT1\\ & \quad\quad \wedge ReturnType(M,X)\wedge Abstract(M)))))))\end{array}$$

To see how this works, suppose that we have the following class:

```
class A {
  abstract T m();
}
```

Converting it to FOL formula yields the following formula.

$$\begin{array}{ll}B = & \forall O(Class(A)\wedge(O \in object(A)) \rightarrow\\ & \quad (Instance(O,A)\wedge Method(m)\wedge Member(m,O)\\ & \quad \wedge ReturnType(m,T)\wedge Abstract(m))))\end{array}$$

A natural deduction proof using set theoretic axioms shows that

$$B \rightarrow A$$

5.4.2 Example 2: Cross referencing between two classes

Next, we will show an example of cross referencing.

Suppose that we have the following classes

```
A1 {
  abstract T m1();
}
A2 {
  abstract A1 m2();
}
```

We convert translated formulae to FOL formulae and paramodulate all equalities; simplifying the set constraints yields the following formula. We can make it easier to read by renaming the variables and further simplify the set constraints by discarding unnecessary constraints. Note that these constraints should not be discarded if they word to be exported, but since we are considering only one pattern now, they can be discarded without affecting the semantics.

$$\begin{array}{ll}(F =) & \exists T\exists s1\exists s2\exists O1\exists O2\\ & \exists s3\exists s4\exists X1\exists X2(\\ & \quad (Class(s1)\wedge(O1 \in object(s1)) \rightarrow\\ & \quad\quad (Instance(O1,s1)\\ & \quad\quad \wedge Method(s3)\wedge Member(s3,O1)\wedge X1 \in T\\ & \quad\quad \wedge ReturnType(s3,X)\wedge Abstract(s3)))\\ & \quad \wedge(Class(s1)\wedge(O2 \in object(s1)) \rightarrow\\ & \quad\quad (Instance(O2,s2)\\ & \quad\quad \wedge Method(s4)\wedge Member(s4,O2)\wedge X2 = s1\\ & \quad\quad \wedge ReturnType(s4,X2)\wedge Abstract(s4))))\end{array}$$

Suppose that we have classes which looks exactly the same as the pattern definition, but is program code. Converting it to FOL formula yields the following formula, where unquantified symbols are constants.

$$\begin{array}{ll}(G =) & \forall O1\forall O2(Class(A1)\wedge(O1 \in object(A1)) \rightarrow\\ & \quad (Instance(O1,A1)\wedge Method(m1)\\ & \quad \wedge Member(m1,O1)\wedge ReturnType(m1,T)\\ & \quad \wedge Abstract(m1)))\\ & \quad \wedge(Class(A2)\wedge(O2 \in object(A2)) \rightarrow\\ & \quad (Instance(O2,A2)\wedge Method(m2)\\ & \quad \wedge Member(m2,O2)\wedge ReturnType(m2,A1)\\ & \quad \wedge Abstract(m2))))\end{array}$$

By natural deduction using set theoretic axioms, we can proof that $G \rightarrow F$

The pattern defined above can also match the following program code.

```
A {
  abstract A m();
}
```

which, when converting it to FOL formula, yields the following formula, where unquantified symbols are constants.

$$
\begin{aligned}
(G' =) \quad & \forall O(Class(A) \wedge (O \in object(A)) \rightarrow \\
& (Instance(O, A) \wedge Method(m) \\
& \wedge Member(m, O) \wedge ReturnType(m, A) \\
& \wedge Abstract(m))))
\end{aligned}
$$

With a similar proof technique, we can proof that $G' \rightarrow F$

5.4.3 Example 3: Method invocation

Suppose that we have the following pattern definition

```
A1 {
  abstract ... m1();
}
A2 {
  A1 f1;
  ... m2() {
    A1 v1, A2 v2 {
      v2=this;
      v1=v2.f1;
      v1.m1();
    }
  }
}
```

After all singleton constraints are simplified, ignoring the return types of the methods, and converted to FOL formula, the formula looks like the following. We make it easier to read by renaming the variables.

$$
\begin{aligned}
(F =) \quad & \exists A1 \exists A2 \forall O1 \forall O2 \\
& \exists m1 \exists f1 \exists m2 \exists v1 \exists v2(\\
& \quad Class(A1) \wedge (O1 \in object(A1) \rightarrow \\
& \quad\quad (Instance(O1, A1) \\
& \quad\quad \wedge Method(m1) \\
& \quad\quad \wedge Member(m1, O1) \wedge Abstract(m3))) \\
& \quad \wedge Class(A2) \wedge (O2 \in object(A1) \rightarrow \\
& \quad\quad (Instance(O2, A2) \\
& \quad\quad \wedge Method(m2) \\
& \quad\quad \wedge Member(m2, O2) \\
& \quad\quad \wedge Type(v1, !A1) \wedge Type(v2, !A2) \\
& \quad\quad\quad \wedge v2 = O2 \wedge m2 <_{use} v2 \\
& \quad\quad\quad \wedge (O2 = v2 \rightarrow \\
& \quad\quad\quad\quad v1 <_{update} f1|_{m2} \wedge m2 <_{use} f1, v2 \\
& \quad\quad\quad \wedge (O1 = v1 \rightarrow \\
& \quad\quad\quad\quad m2 <_{invoke} m1|_{m2} \wedge m2 <_{use} m1, v2))))
\end{aligned}
$$

Suppose that we have classes, which looks exactly the same as the pattern definition, but is program code. Converting it to FOL formula yields the following formula, where unquantified symbols are constants.

$$
\begin{aligned}
(G =) \quad & \forall O1 \forall O2(\\
& \quad Class(A1) \wedge (O1 \in object(A1) \rightarrow \\
& \quad\quad (Instance(O1, A1) \\
& \quad\quad \wedge Method(m1) \\
& \quad\quad \wedge Member(m1, O1) \wedge Abstract(m3))) \\
& \quad \wedge Class(A2) \wedge (O2 \in object(A1) \rightarrow \\
& \quad\quad (Instance(O2, A2) \\
& \quad\quad \wedge Method(m2) \\
& \quad\quad \wedge Member(m2, O2) \\
& \quad\quad \wedge Type(v1, !A1) \wedge Type(v2, !A2) \\
& \quad\quad\quad \wedge v2 = O2 \wedge m2 <_{use} v2 \\
& \quad\quad\quad \wedge (O2 = v2 \rightarrow \\
& \quad\quad\quad\quad v1 <_{update} f1|_{m2} \wedge m2 <_{use} f1, v2 \\
& \quad\quad\quad \wedge (O1 = v1 \rightarrow \\
& \quad\quad\quad\quad m2 <_{invoke} m1|_{m2} \wedge m2 <_{use} m1, v2))))
\end{aligned}
$$

By natural deduction using set theoretic axioms, we can proof that $G \rightarrow F$

The pattern defined above can also match the following program code.

```
A1 {
  abstract ... m1();
}
A2 extends A1 {
  A1 f1;
  ... m1 overrides A1.m1() {
    A1 v1, A2 v2 {
      v2=this;
      v1=v2.f1;
      v1.m1();
    }
  }
}
```

The next example shows how to distinguish these two source code.

5.4.4 Example 4: Directives

We modify the pattern in Example 3 by changing the method invocation statement.

```
inSuperType v1.m1();
```

The directive indicates that `v1` must have a type that is a super type of `A1`. Therefore, if we feed the two program listed in Example 3, only the latter will be matched by this pattern. If we want to match only the former, we can write two statements one with the not statement, another without, by changing the method invocation statement.

```
not inSuperType v1.m1();
v1.m1();
```

6. DISCUSSION

We can add a little syntactic sugar to the core language to make it easier to read and write. We can define the translation of the simplify the syntax to the core language by the following rule. We denote the normalization relation by $A \rightsquigarrow B$. The Singleton Set rule eliminates the requirement that even singleton sets have to be followed by the selection constructs when in an applicational position. We allow multiple local variable definitions be compressed into one using the Local Variables rules. We can also use low dots to generate fresh variables.

The basic forms of statements are limited. It is tedious to write pattern definition in the basic forms of statements as shown in the example. We introduce several constructs to address this problem. We use $\mathcal{E}$ to denote the term with a hole in it and $\mathcal{E}[X]$ to be the term to be translated. The rules in Figure 10 should be applied repeated until the syntax form is in the PsPDL forms.

Combining these rules, we can rewrite the pattern definition given in Example 3 as

```
A1 {
  abstract ... m1();
}
... {
  A1 f1;
  ... m2() {
    this.f1.m1();
  }
}
```

$$\frac{n \text{ singluar} \quad X \text{ fresh} \quad n \text{ in an applicational position other than extends or overrides}}{n \rightsquigarrow n[\textbf{some } X]} \text{(Singleton Set)}$$

$$\frac{}{t_1\ n_1; t_2\ n_2; \ldots; t_w\ n_w\{s\} \rightsquigarrow \\ t_1\ n_1\{t_2\ n_2\{\ldots\{t_w\ n_w\{s\}\}\ldots\}\}} \text{(Local Variables)}$$

$$\frac{x \text{ fresh}}{\ldots \rightsquigarrow x} \text{(Omission)}$$

$$\frac{v \text{ fresh}}{\mathcal{E}[\textbf{this}] \rightsquigarrow \\ \ldots\ v\{v = \textbf{this}; \mathcal{E}[v]\}} \text{(Self)}$$

$$\frac{v \text{ fresh}}{\mathcal{E}[\textbf{super}] \rightsquigarrow \\ \ldots\ v\{v = \textbf{super}(t); \mathcal{E}[v]\}} \text{(Superobject)}$$

$$\frac{\overline{v} \text{ fresh}}{\mathcal{E}[(\overline{X})] \rightsquigarrow \\ \ldots\ \overline{v}\{\overline{v} = \overline{X}; \mathcal{E}[\overline{v}]\}} \text{(Parameters)}$$

Figure 10: Translation of Extended Syntax

Efficiency is another issue that is usually considered when implementing an algorithm that discover patterns in source code. The formulae generated from the PGRs can be quite inefficient to handle by general purpose theorem provers because a lot constraints on singleton sets are not simplified. We can simplify the constraints using a technique that is often referred to as paramodulation in theorem proving literature, and discard the constraints when the simplification is accomplished. This will result in much simplified formulae which we have demonstrated in the examples. Also, handling set theoretic axioms is much faster for some theorem provers that have specialized strategies for set theoretic formulae. These theorem provers usually have switches for formulae involving set theoretic axioms that can be turned on to handle the queries used in our approach.

We have a translator of PsPDL to TPTP format[13, 12] using ANTLR and Java so that the generated FOL formulae can be read by theorem provers like Prover9 (which is the successor of Otter), Vampire, Darwin, and OSHL, among others. The previous work on SPQR provides a very good modeling tool for C++ programs into both POML and input for Otter. Other filters can be written to translate from programs written in other programming languages to POML and translating POML to TPTP format.

7. REFERENCES

[1] Smith, J., and D. Stotts, "Intent-oriented Design Pattern Formalization using SPQR," Design Pattern Formalization Techniques, T. Taibi, ed., IGI Publishing, 2007, pp. 123-155.

[2] Smith, J., and D. Stotts, "SPQR: Flexible Automated Design Pattern Extraction from Source Code," Proc. of Automated Software Engineering 2003, Montreal, Oct. 6-10, 2003, pp. 215-224.

[3] Herranz, A., J.J. Moreno and N. Maya, "Declarative Reflection and its Application as a Pattern Language," WFLP 2002 11th Intl. Workshop on Functional and (Constraint) Logic Programming, in Electronic Notes in Theoretical Computer Science Volume 76, Nov. 2002, pp. 197-215.

[4] Bünnig, S., P. Forbrig, R. Lämmel, N. Seemann, "A Programming Language for Design Patterns," GI Jahrestagung 1999, pp. 400-409.

[5] Lovatt, H. C., A. M. Sloane, and D. R. Verity, "A pattern enforcing compiler (PEC) for Java: using the compiler," Proc. of the 2nd Asia-Pacific Conf. on Conceptual Modelling - Volume 43, S. Hartmann and M. Stumptner, eds., Conferences in Research and Practice in Information Technology Series, vol. 107, Australian Computer Society, Darlinghurst, Australia, 2005, pp. 69-78.

[6] Eden, A., E. Gasparis, J. Nicholson, "LePUS3 and Class-Z Reference Manual," Dept. of Computer Science, University of Essex, Tech. Rep. CSM-474, ISSN 1744-8050, 2007.

[7] Kramer, C. and L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software," Proc. of the 3rd Working Conference on Reverse Engineering (WCRE '96), IEEE Computer Society, Washington, DC, 1996, 208.

[8] Niere, J., W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," Proc. of the 24th Intl. Conference on Software Engineering, ICSE '02, ACM, New York, NY, 2002, pp. 338-348.

[9] Pettersson, N. "Measuring precision for static and dynamic design pattern recognition as a function of coverage," Proc. of the 3rd Intl. Workshop on Dynamic Analysis, ACM, New York, NY, 2005, pp. 1-7.

[10] Bosch, J., "Design Patterns as Language Constructs," JOOP 11(2), 1998, pp 18-32.

[11] ISO/IEC 13211: Information technology — Programming languages — Prolog. International Organization for Standardization, Geneva.

[12] Sutcliffe, G. and C. B. Suttner, "The TPTP Problem Library: CNF Release v1.2.1," Journal of Automated Reasoning, Volume 21, 2, 1998, pp. 177-203.

[13] "SyntaxBNF", http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html.

[14] Woolf, B., "The object recursion pattern", Pattern Languages of Program Design 4, N. Harrison, B. Foote, and H. Rohnert, eds., Addison-Wesley, 1998.

[15] Zimmer, W., "Relationships between design patterns," Pattern Languages of Program Design, Coplien and Schmidt, eds., Addison- Wesley, 1995, pp 345–364.

[16] Xu, H., "FOLSVI: First Order Logic with Strict Variables and Indices," Tech. Report, Computer Sci. Dept., Univ. of North Carolina at Chapel Hill, 2008.

[17] Abadi, A. and L. Cardelli, "A Theory of Objects," Monographs in Computer Science (Gries and Schneider eds.), Springer-Verlag, New York, 1996.