FOLIE: A FIRST ORDER LOGIC INTERFACE AND EVALUATOR
COEN 260

TIM DION

Abstract. In very large Object Oriented systems it becomes difficult to manage complex dependencies between interconnecting modules. The authors of this paper proposes using expressions in First Order Logic[3][2] to reason about module prerequisites and class relations. Specifically, we propose a First Order Logic expression evaluator to make queries in a live, run time environment.

1. Introduction

As a part of a very large software porting project, the authors have been tasked with converting a Smalltalk [4] [5] based system [6] to a new version of the Visual-Works Smalltalk language and IDE. Through our initial proof of concept port we seen several types of problems that require minor code changes and fixes. The most common problem emerges from failures in establishing load order of modules and their dependents.

We extrapolated the time required to port the proof of concept GUI application over the entire system. We ported roughly 8 modules in one month. We estimated that porting the entire system (of some 78 modules) would take six months. Clearly we needed some type of automated tool to find and fix these module dependency problems. We estimated that we could save two man months through an automated porting solution.

The tool we developed is called FOLIE, it is a First Order Logic expression interpretor for Smalltalk. FOLIE gives us the ability to make queries against a run time system to find missing module prerequisites and module dependency collisions. For now, a module can be defined as a group of Classes, rather like a Java Package ¡add ref¿. Further on in this document we will present a more detained explanation of Smalltalk modules and mismapping issues between source code repositories.

In this paper we present a detailed analysis of the module issues in porting a very large Smalltalk application. We will then give a quick introduction to First Order Logic and some of its principles. We will derive a few useful expressions and show how then can formally state various properties of our system. Finally, we will talk a bit about the implementation of FOLIE.

Before we get to deep into the details, it is important to point out that FOLIE is not a Theorem Solver. It does not grind through premises to determine if conclusions logically follow. FOLIE evaluates propositions to return a true or false result. This evaluation takes places over the domain of Classes in the target Smalltalk system. However, if one thinks of a Smalltalk runtime environment as as an axiomatic

Date: Fall 2007.

system, then FOLIE could (with much hand waving) be though of as a kind of empirical theorem solver.

Finally it is important to point out, the results and implementation of FOLIE need not be specific to Smalltalk and its run time environments. Indeed, First Order Logic makes an especially good query language because it is highly generalized. If we have been careful with the definitions of our predicates, we could to port FOLIE to any Object Oriented Run time system with sufficient reflexivity.

2. The Problem

In this section we dig into the specifics of Smalltalk, IDEs, and Source Control module structures. We will try to explain the problem we face in detail. It is worth noting that the problems explained here can be generalized to other Object Oriented development environmental.

As with Java Packages, Smalltalk can be modularized into components. However, these components were added on to Smalltalk to facilitate source code control. The first (and most successful) source control system was called Envy/Developer [10] by Object Technology International (OTI), from Toronto Canada. Envy was far superior to any source control technology at the time. IBM purchased the technology an incorporated it into VisualAge for Java ¡add refer¿. Unfortunately, IBM hitched its wagon to The Eclipse Java IDE. They decided to phase out the VisualAge products, which also orphans support for the OTI Smalltalk Envy products.

For these reasons, the authors are porting a Cincom VisualWorks Smalltalk ¡addref¿ application from Envy to a new source code control technology called Store from Cincom. However, A complex set of problems emerge in how module dependents are managed between the two systems.

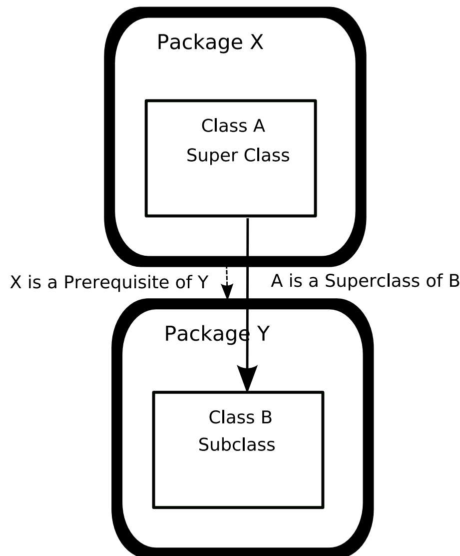

Figure 1. Package Dependencies

We see a category of problems where the chains of module prerequisites have not been well defined in the source environment. It bares noting that Envy uses

structures called Configuration Maps and Applications to structure source code a modularize the system. For the sake of this papers, we will use the generic term *Package* to refer to these structures.

In Figure 1 we can see an illustration of the problem. Class B subclasses Class A. Therefore, Class A requires that Package X has been loaded before we can load Package. We would expect that Package Y has an explicitly define prerequisite in its package definition.

Any system that growths organically over several years will subject to a certain amount of untidiness and entropy. In the case of our existing legacy many of these explicitly defined prerequisites have not been created. It turns out that Envy has mechanisms to manage undefined class references. Envy's more permissive model meant that software engineers did not have to define every package prerequisite.

In moving code to the Store environment, we have not been so lucky. Store does not manage dependents the same way as Envy. Store is not permissive in the way it loads Packages [1]. It has become clear that we need a query tool to find these types of problems. This tool should be flexible and powerful enough to describe any type of relation in the Smalltalk environment, not just the given example above. At the same time, this tool should be able to formally describe dependency properties mathematically and concisely. We believe First Order Logic is sufficiently descriptive for this task.

3. First Order Logic

In this section we give a short, simple, and non-rigorous introduction to First Order Logic. We hope to give enough background on First Order Logic so the reader can understand statements made in FOLIE.

But lets start with the more familiar Propositional Logic. Propositions are statements that evaluate to true or false. Atomic Propositions are statements which we can not decompose into further propositions. The statement, "Do you walk to school or buy your lunch?" is an infamous elementry school riddle. We can model this statement in Propositional Logic as: $\mathsf{P} \lor \mathsf{Q}$. In this case P stands for, *Do you walk to school?* and Q stands for, *Do you buy your lunch?*. Here P and Q are atomic propositions.

Formulas are compound propositions built using operators using conjunction, disjunction, negation and conditional symbols $\{\land, \lor, \neg, \rightarrow\}$. We can inductively define a formal semantics [2] with the following definitions[9][8].

If φ is a Formula then one of the following must hold:

(1) φ is an atomic;
(2) φ is $\neg\psi$ is a formula.
(3) φ is $\psi \land \chi$ for some unique φ and ψ;
(4) φ is $\psi \lor \chi$ for some unique φ and ψ;
(5) φ is $\psi \rightarrow \chi$ for some unique φ and ψ;

Some definitions of Propositional Logic use the $\perp$ symbol to represent contradictions. We currently do not include a means to describe contradictions in FOLIE. Deductive systems require a semantics for contradictions in order to describe things like Proof by Contradiction. At some point in future versions of FOLIE we may add

[1] It should be noted that Store has significant advantages over Envy in many areas.
[2] One may think of an inductive definition like a Backus-Naur Form grammar structure.

contradiction symbols so we can reason about the consistency of Classes, Packages, and Prerequisites.

3.1. **FOL.** Now, on to First Order Logic. We will start with a few examples and then reconstruct them to get a sense of the semantics. We wont mount a champaign to give a formal definition of First Order Logic in this paper. This would be a considerable undertaking. See for [3] for a clear formal introduction to First Order Logic.

We use sentences from the pedagogic software called Tarski's World[1] to give a rough idea how FOL works.

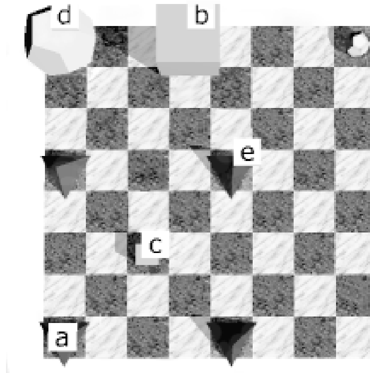

FIGURE 2. Tarski's World

Tarski's World teaches logic by placing objects onto a chess board. Objects can be tetrahedra, cubes, or dodecahedra. They can be small, or large. FOL [3] uses predicate expressions to describe relations between things in a world (domain). Note, in figure 2 we see labels *a, b, c, d* and *e.* We can make simple propositions with built in predicates in Tarski's World. such as:

Small(a) Larger(a,b) SameShape(a,c) Dedec(c)

Labeled objects are referenced directly, but what if we want to say useful things about sets of objects or make general statements? We do this by introducing variables to the language, but we have to be explicit about what the variables stand for. We can't say something like Small(x) without binding x to objects in the domain. In other words the statement Small(x) is not a proposition because it does not evaluate to a true or false value. If we add quantifiers to this statement then we can build propositions. It turns out we only need the universal quantifier $\forall$ and the existential quantifier $\exists$ to say almost everything useful about things that live in a domain [4]. A few examples follow:

$$\forall x[\mathsf{Small}(x)]$$

The expression above says: All things are small.

[3]We will use the abbreviation *FOL* to refer to First Order Logic.

[4]This statement is false. FOL is not expressive enough to define and reason about structures like relations as first class objects. FOL also can not express the property of finiteness. FOL does, however, generally express many things useful to Software Engineers.

$$\exists x \exists y \mathsf{Larger}(x,y)$$

And this expression says: There exists at least one thing that is larger than something else.

FOL takes all the same operators from Propositional Logic $\{\wedge, \vee, \neg, \rightarrow\}$. We also add equivalence and nonequivalence $\{=, \neq\}$. These operators can not occur within the terms of predicates. Many definitions of FOL include the biconditional operator $\{\leftrightarrow\}$. Biconditionals are really just syntaxic sugar for complex operators[5], so we won't include them here.

Table 1 shows an example form the literature (page 300, Language, Logic and Proof) [2]. These example statements are all true in the Tarski's world pictured in Figure 2. These examples show that one can build expressive and complex statements in FOL. As we will see later, FOL can express a variety of claims about software structures.

1. Every tetrahedron is in front of every dodecahedron. $\forall x[\mathsf{Tet}(x) \rightarrow \exists y[\mathsf{Dodec}(y) \rightarrow \mathsf{FrontOf}(x,y)]]$
2. No dodecahedron has anything in back of it. $\neg\exists x\, [\mathsf{Dodec}(x) \wedge \exists y[\mathsf{BackOf}(y,x)]]$
3. No tetrahedron is the same size and as any cube. $\neg\exists x[\mathsf{Tet}(x) \wedge \exists y[\mathsf{SameSize}(x,y) \wedge \mathsf{Cube}(y)]]$
4. Every Dodecahedron is the same size as some cube. $\forall x\, [\mathsf{Dodec}(x) \rightarrow \exists y[\mathsf{SameSize}(y,x) \wedge \mathsf{Cube}(y)]]$
5. Anything between two dodecahedra is a cube. $\forall x[\exists y \exists z(\mathsf{Dodec}(y) \wedge \mathsf{Dodec}(z) \wedge \mathsf{Between}(x,y,z)) \rightarrow \mathsf{Cube}(x)]$
6. Every cube falls between two objects. $\forall x[\mathsf{Cube}(x) \rightarrow \exists y \exists z[\mathsf{Between}(x,y,z)]]$
7. Every cube with something in back of it is small. $\forall x[(\mathsf{Cube}(x) \wedge \exists y[\mathsf{BackOf}(y,x)]) \rightarrow \mathsf{Small}(x)]$
8. Every dodecahedra with nothing to its right is small. $\forall x[(\mathsf{Dodec}(x) \wedge \neg\exists y[\mathsf{RightOf}(y,x)]) \rightarrow \mathsf{Small}(x)]$
9. Every dodecahedron with nothing to its right has something to its left. $\forall x[(\mathsf{Dodec}(x) \wedge \neg\exists y[\mathsf{RightOf}(y,x)]) \rightarrow \exists z[\mathsf{LeftOf}(z,x)]]$
10. Any dodecahedron to the left of a cube is large. $\forall x[(\mathsf{Dodec}(x) \wedge \exists y[\mathsf{Cube}(y) \wedge \mathsf{LeftOf}(x,y)]) \rightarrow \mathsf{Large}(x)]$

TABLE 1. FOL Example for Tarski's World.

As a better example, if we invent predicates B(something) and A(something) we can use FOL to model the classic Aristilian Propositions[7]. Note, the translations from English to FOL don't always work well. We define predicate B(something) as true when the *something* has property B.

4. NOTES ON IMPLANTATION

FOLIE uses SmaCC, a Compiler Compiler for Smalltalk developed by the Smalltalk Refractory. The following lists the BNF in SmaCC format:

[5] We won't give a proof here, but it can be shown that: $\mathsf{P} \leftrightarrow \mathsf{Q} \equiv \mathsf{P} \rightarrow \mathsf{Q} \wedge \mathsf{Q} \rightarrow \mathsf{P}$

A	All A are B	$\forall x[A(x) \to B(x)]$
E	All A are not B	$\forall x[A(x) \to \neg B(x)]$
I	Some A are B	$\exists x[A(x) \land B(x)]$
O	Some A are not B	$\exists x[A(x) \land \neg B(x)]$

TABLE 2. Aristilian Propositions

```
formula : disjunction "->" formula
| disjunction;

disjunction : conjunction "|" disjunction
| conjunction;

conjunction : negation "&" conjunction
| negation;

negation : "~" negation
| general;

general : quantifier negation
| atom;

atom : atomic_formula
| "(" formula ")"
| "[" formula "]";

quantifier : "_V"  variable
|  "_E" variable;

atomic_formula :  relation_symbol
| relation_symbol "(" term ")"
| relation_symbol "(" term "," term ")"
| term "=" term
| term "~=" term;

term : variable
| literal;

relation_symbol  : <relation_symbol>;
variable : <variable>;
literal : <literal>;
```

We supplied the scanner with the following definitions.

```
<variable> : [w-z];
<literal>: \#[a-zA-Z0-9\-\ ]* ;
<relation_symbol>  : [TF] | [A-Z][a-zA-Z]*;
<whitespace>    :       \s+;
```

Converting well formed expression in First Order Logic symbols into simple Unicode text took some creativity. To differentiate literal Object references we prefix them with a pound symbol. Smalltalk developer will find this style of references familiar. We add pseudo relations symbols called "T" and "F". These are really just predicates that take no parameters. As expected, they evaluate to true and false. These we implemented to make unit testing easier. We only allow variables w,x,y, and z.

Literals can contain blank spaces. Unfortunately. Store allows spaces in Package names. We could have required quotes around symbols with spaces, as done in Smalltalk, but this mechanism caused other parsing issues. We created an interim solution to use the parser tokens to delineate the end of a literal. Regrettably this means that putting spaces at the end of literals becomes problematic. We will revisit this functionality in the future.

FOLIE	FOL	Description
#Object	label	Smalltalk class or Package.
$->$	$\rightarrow$	Conditional
$\vert$	$\vee$	Disjunction
&	$\wedge$	Conjuction
˜	$\neg$	Negation
$_V$	$\forall$	Universal Quantifier
$_E$	$\exists$	Existential Quantifier

Table 3. FOL to text translations

Predicates must start with a capital character. All predicates are predefined as Smalltalk methods. Currently there are four predicates implemented. Adding additional predicates requires changing two methods.

Predicate	Meaning
Subclass($term_1$, $term_2$)	true if $term_1$ is a subclass of $term_2$.
Packages($term_1$, $term_2$)	true if $term_1$ is a Package that contains $term_2$.
References($term_1$, $term_2$)	true if $term_1$ is a class that references class $term_2$.
Prerequisite($term_1$, $term_2$)	true if $term_1$ is a Package required by $term_2$.
Show($term_1$)	always true, prints $term_1$ to transcript.

Table 4. Predicates

4.1. **Node Trees.** While the code is being parsed the SmaCC system builds Smalltalk objects for each parse node. The end result creates a tree of objects which represent an expression's structure. We trigger computations in FOLIE by evaluating string expressions like:

```
'_Vx[Subclass(x, #Object) -> (x ~= #Object) ]' eval.
```

The tree in figure 3 shows the generated structure of the expression above. A FOLBinding object holds the entire tree. When we send this object the *eval* method it tells the quantifier to map the domain (hard coded to be all the Smalltalk classes and Packages) onto the quantifiers bond variable references. This means the FOLQuant object searches the tree and binds replace variable references with domain objects for each iteration over the domain. Next *eval* gets passed down the chain to a

FOLOperator object, which sends *eval* to its sub objects and then performs its required boolean operation. This recursive computation returns a boolean value, once it unwinds up the stack.

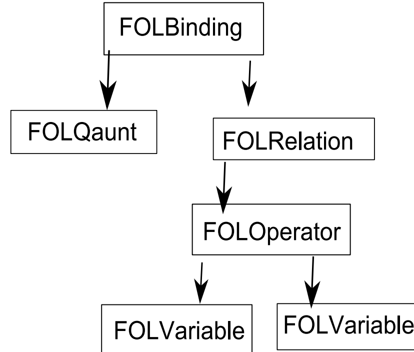

FIGURE 3. Node Tree

It seems like it should have been possible to build a parse tree using Smalltalk Block Closures. Block Closures in Smalltalk provide a semantics for Lambda like expressions which are applied over a set of objects. However, all the correct bits of information are not available at parse time to simply insert values into prebuilt closures. For example, we can not insert a subnode into a structure when we don't know the type of those subnodes before hand. This problem necessitated building intermediate objects to represent parse tree structures.

5. Discussion of Results

Now we get to the interesting part. What did we find out.

This very simple proposition just prints out all the classes `x` such that `x` is contained by the Package called #Logic. The Show() predicate turned out to be extremely useful for understanding what was happening inside expression evaluations. By using the Show(x) predicate we can incrementally build a sentence that expresses the consistency of a Package. Formally, sentences are expressions with no free variables, that is variables not bound by a quantifiers ∀ and ∃.

```
'_Vx[Packages(#Logic, x) ->  Show(x)]' eval
```

Next, building our expression on #Logic, we find every object `x` contained in Package #Logic, is Package `y` of the superclass of `x` a prerequisite of #Logic.

```
'_Vx[Packages(#Logic, x) ->  _Vy[Subclass(x,y) -> Show(y)]]' eval
```

Now we find all prerequisite Packages defined implicitly through inheritance.

```
'_Vx[Packages(#Logic, x) ->  _Vy[Subclass(x,y) ->
_Vz[Packages(z,y)->Show(z) ]]]' eval
```

Finally, we test that our implicit prerequisites are proper explicitly defined Package prerequisites. In this case, the Package called #Logic is inconsistent. It does not have well formed Prerequisites. Note, we could extend this sentence with another quantifier to test if all Packages are inconsistent or that there does not exist an inconsistent Package.

```
'_Vx[Packages(#Logic, x) ->  _Vy[Subclass(x,y) -> _Vz[Packages(z,y)->
(Prerequisite(y, #Logic) & Show(y))]]]' eval
```

5.1. **OOP.** We began to realize at some point that we could make interesting assertions on the nature of Object Oriented Programming. In fact, we can begin to isolate properties that define Object Orientation. This next proposition says that all objects should be subclasses of Object, with the exception of Object itself.

```
'_Vx[Subclass(x, #Object) -> (x ~= #Object) ]' eval.
```

Given more time, this line of research on formal OOP properties would generate interesting results. We would like to have reasoned more about polymorphism, metaclasses, and object prototyping.

6. Future Work

Now, this seems like a lot of work and thought to perform a task that could have been done with a few well crafted Smalltalk methods. But there are a few reasons why we feel the effort has been well spent.

Firstly, we now have a formal FOL semantics and evaluation engine to reason about general problems. We can extend our predicates to talk about UML structures, aggregate object sets, formalized type systems, and anything else that lives in the domain of a Smalltalk system.

Secondly, we have formalism. We can copy and paste our sentences directly into documents like requirement specifications. We can execute formal specifications described in FOL to verify system constraints and assertions. We can prove sentences empirically true for system properties.

Finally, and most importantly, we have a platform to build more complex behaviors and documentation systems. For future work, we can do things like auto generate provably true documentation systems. The authors are especially intrigued with design warehouses. We could build a self aware (i.e. as in reflexive not as in consciousness) tool that stores design information. For example, we could generate system architecture descriptions from such a database. FOLIE creates a new set of options for reasoning about and expressing properties in run time systems in Smalltalk.

References

[1] Jon Barwise and John Etchemendy. *The Language of First-Order Logic (Windows Program, Tarski's World), 3rd Ed., Revised & Expanded*. CSLI Publishing, 1993.

[2] Jon Barwise and John Etchemendy. *Language, Proof and Logic*. Center for the Study of Language and Inf, April 2002.

[3] H. Enderton. *A Mathematical Introduction to Logic*. Academic Press, London, UK, 2002.

[4] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[5] Adele Goldberg and David Robson. *Smalltalk 80 : The Language*. Addison-Wesley Series in Computer Science. Addison-Wesley Professional, January 1989.

[6] T. Hawker. On the development of a platform- and domainneutral enterprise application integration framework. In *Smalltalk Solutions 2006, Toronto, 24-26 May 2006*. Smalltalk Solutions, 2006.

[7] Patrick J. Hurley. *Logic, A Concise Introduction*. Wadsworht, 2000.

[8] Bob Riemenschneider. Coen 385: Formal methods for software engineering lecture notes. http://www.csl.sri.com/users/rar, January 2003.

[9] Bob Riemenschneider. Coen 260: Truth, deduction, and computation, lecture notes. http://www.csl.sri.com/users/rar/logic-course.html, November 2007.

[10] J. Steinman and B. Yates. Object technology's envy/developer. *The Smalltalk Report*, October 1992. http://www.bytesmiths.com/Publications/9209Envy.html.