6.825 Project 1b

Due: Wednesday, October 16, 5 PM

October 4, 2002

One major concern of software engineers is the validation and verification of software. Given a piece of software, and a specification of what it's supposed to do, how can we be sure that it really does what it's supposed to do? Since software programs are formal objects, not entirely unlike sentences of a logic, it is appealing to try to use logical methods to *prove* that a program satisfies its specification.

In general, the kinds of reasoning one needs to do about a software program are first-order, rather than propositional. You need to talk about all values of a particular variable, or all states of a data structure, which is very cumbersome in propositional logic. So, to prove programs correct, we need to use first-order logic theorem proving. And, although it's certainly possible, in principle, for computers to do proofs in first-order logic, the process tends to scale very badly, making it virtually impossible to do any but the simplest proofs autonomously.

An alternative approach, taken by Prof. Daniel Jackson and his research group here at MIT (`http://sdg.lcs.mit.edu/alloy`), is to make the problem somewhat easier for the computer by being slightly less ambitious about what we're asking it to show.

As you all know, there are reasonably effective algorithms for computing whether a propositional sentence is satisfiable. So, if we can formulate our program verification problem as a SAT problem, we might have some leverage on it.

1 Working in a finite domain

Because the program verification problem is fundamentally first order, we can't solve it exactly using SAT. But what we *can* do is restrict the first-order problem to a small universe. If we have a finite universe, we can convert any first-order logic theory into a propositional logic theory by eliminating quantifiers.

Given a finite universe $U = \{u_1, \ldots, u_n\}$, and letting $\varphi(x)$ stand for an arbitrary logical expression containing variable x, we can rewrite the sentence

$$\forall x . \varphi(x)$$

as

$$\varphi(u_1) \wedge \cdots \wedge \varphi(u_n) \quad ,$$

and the sentence

$$\exists x . \varphi(x)$$

as

$$\varphi(u_1) \vee \cdots \vee \varphi(u_n) \ .$$

So, for example, if our domain were $U = \{a, b, c\}$, we could rewrite

$$\forall x. \exists y. r(x, y)$$

as

$$(r(a, a) \vee r(a, b) \vee r(a, c)) \wedge (r(b, a) \vee r(b, b) \vee r(b, c)) \wedge (r(c, a) \vee r(c, b) \vee r(c, c)) \ .$$

Back to the software example, imagine that we want to prove a file-locking system correct. Although it's too hard to do that in general, we might focus our attention on a finite domain that contains m files and k users.

We could then use logical statements to describe the system and to assert that some file is corrupted. If this conjunction of logical statements is *satisfiable*, then we have found a bug in the program, and the satisfying assignment will tell us which particular assignment of values to variables causes the problem.

If the conjunction is *unsatisfiable*, then we have shown that the system behaves correctly in this particular small domain; note that we have not shown it to be bug free in general—it might require some larger number of files and users to reveal the bug.

So, rather than proving programs correct, we'll work on revealing bugs in small instances of the program. The focus on finding bugs means that we can usefully employ SAT algorithms that are incomplete, such as WalkSAT (as well as complete methods, such as DPLL).

2 Train domain

For this project we will focus on some simple protocols for avoiding collisions between trains in a railroad network. We will describe the domain and the protocols using first-order logic. As in the situation calculus, we will *reify* situations, using variables that range over possible arrangements of trains on the tracks.

We'll assume that the train network is divided into segments (called "rails") that are connected to one another in a directed network. So, for example, we can describe the one-way loop layout in figure 1 using the expression

$$\mathit{connects}(R_1, R_2) \wedge \mathit{connects}(R_2, R_3) \wedge \mathit{connects}(R_3, R_4) \wedge \mathit{connects}(R_4, R_1) \ . \tag{1}$$

An individual *connects* statement implies that the rails are oriented and connected in such a way that a train can move from one to the next. We will refer to this configuration of rails as layout 1. We will also need to assert that there are no other connections:

$$\begin{aligned} &\neg\mathit{connects}(R_1, R_3) \wedge \neg\mathit{connects}(R_1, R_4) \wedge \neg\mathit{connects}(R_2, R_4) \wedge \neg\mathit{connects}(R_2, R_1) \wedge \\ &\neg\mathit{connects}(R_3, R_1) \wedge \neg\mathit{connects}(R_3, R_2) \wedge \neg\mathit{connects}(R_4, R_2) \wedge \neg\mathit{connects}(R_4, R_3) \wedge \\ &\qquad\qquad\qquad\qquad\qquad\qquad \forall r. \neg \mathrm{connects}(r, r) \ . \end{aligned} \tag{2}$$

We will use $\mathit{on}(t, r, s)$ to mean that train t is on rail r in situation s, and we will say that a situation is safe if and only if no two trains occupy the same rail:

$$\forall s. \mathit{safe}(s) \leftrightarrow \forall r, t_1, t_2. (\mathit{on}(t_1, r, s) \wedge \mathit{on}(t_2, r, s) \rightarrow \mathit{Equals}(t_1, t_2)) \ . \tag{3}$$

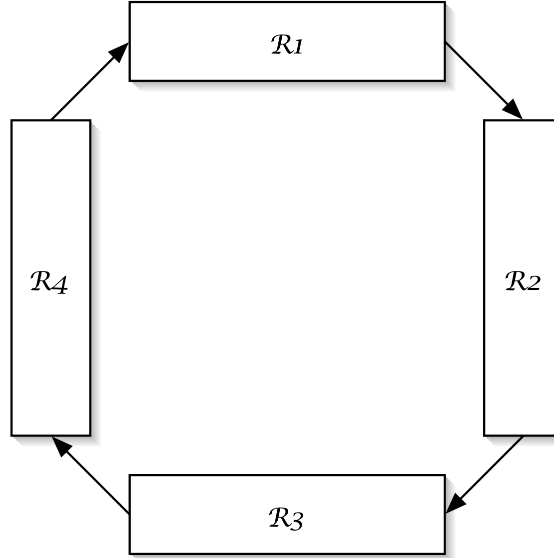

Figure 1: Diagram of layout 1.

If we can show that the one-step dynamics of the trains are safe, it's pretty clear by induction that we can maintain safety from one situation to the next, so that the trains will always be safe. So, we'll just use two constant situations, S_1 and S_2, and we'll assert that S_1 is safe,

$$\mathit{safe}(S_1) \quad . \tag{4}$$

We would like to be able to consider non-deterministic movement protocols for the trains (giving them choices about which way to move), so when we describe the dynamics of the system, we have to allow non-determinism. We can't say exactly where the trains will move, but we can constrain them to follow the laws of physics (that is, only to move to connected rails) and also to follow some laws of convention, which we will specify in the definition of the *legal* predicate. We can formally describe the dynamics, then, as

$$\forall t, r_2 . \mathit{on}(t, r_2, S_2) \rightarrow \exists r_1 . \mathit{on}(t, r_1, S_1) \wedge \mathtt{connects}(r_1, r_2) \wedge \mathit{legal}(t, r_1, r_2) \quad . \tag{5}$$

Now, our job is to say when it is legal for a train to move from one rail to the next, depending only on the current situation (it's not fair to be clairvoyant and look at attributes of the next situation). Here is a candidate definition, which says that it's legal to move to any connected rail that is currently unoccupied:

$$\forall t_1, r_1, r_2 . \mathit{legal}(t_1, r_1, r_2) \leftrightarrow (\neg \exists t_2 . \mathit{on}(t_2, r_2, S_1)) \quad . \tag{6}$$

One of our tasks will be to determine whether this definition of "legal" guarantees that the trains will always be safe.

Ordinarily, we would also need to assert that trains, rails, and situations are exhaustive and mutually exclusive. For the purposes of this assignment, we have built a kind of *ad hoc* type system into the software that converts from first-order to propositional logic assignments, which means we don't need to do this.

In an untyped system, quantified variables range over every element in the universe. In our system, quantified variables only range over elements in the universe with the same type. To keep a simple syntax, we use the following convention to indicate types: A variable ranges over exactly those elements in the universe which share the same, case insensitive, first letter as the variable. Thus, in "all t p(t)", the "t" will only be filled in with elements of the universe beginning with "t" or "T". A good convention would be to use capital letters for constant terms, and lowercase letters for variables.

3 Tasks

1. We'll start work in a domain with two trains, called T_1 and T_2. If we're using layout 1, that makes our whole universe $U = \{T_1, T_2, R_1, R_2, R_3, R_4, S_1, S_2\}$. Write the formulas 1 through 6 in a file using the language of figure 3.
2. You'll now need to add two more axioms to your file, specifying that:
   - Every train is always on some rail; and
   - No train is on two rails at the same time.
3. Convert your whole specification into a CNF sentence, using the converter described in section 4.1. Now, using your DPLL implementation, or the fast one described in the next section, find a satisfying assignment. Where are the trains in S_1 and S_2? If this assignment is unreasonable, you should change your axioms until you get a reasonable assignment.
4. Now, add the assertion that there's a possible train wreck, $\neg safe(S_2)$, and use DPLL to show that there are no possible crashes in this domain. Would it have been okay to use WalkSAT for this job?
5. Modify the definition of *legal* to include allowing the train to stay on the rail it's currently on. You may also need to modify the *connects* relation. Is this domain still safe? Demonstrate with DPLL.
6. Now, let's consider Layout 2, shown in figure 2. Here is part of its logical description (you'll have to fill in the rest):

$$connects(R_1, R_2) \wedge connects(R_3, R_2) \wedge connects(R_1, R_4) \wedge \\ connects(R_3, R_4) \wedge connects(R_2, R_1) \wedge connects(R_4, R_3) \quad . \tag{7}$$

   Demonstrate that it is unsafe, given your definition of *legal* from item 5.

7. Write a new definition of *legal*, which outlaws the bad behavior demonstrated in the previous task. Prove that there will be no crashes in this domain. Show that there are safe configurations for S_1 and S_2. What are they?
8. One way to avoid crashes is never to move. So, we'd also like to show that the system doesn't become *deadlocked*. The system is deadlocked, if there is no train that can move.

   Add an axiom that constrains S_1 to be deadlocked. Using your definition of *legal* from item 7 show that there is no deadlock in layout 1.

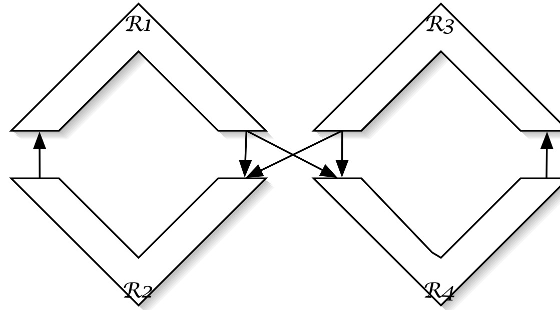

Figure 2: Diagram of layout 2.

9. Show that there can be a deadlock in layout 2. What is the *on* relation in your example?

10. (tricky) Revise your definition of *legal* so that you no longer can have a deadlock in layout 2. (*H*int: you may need to add an extra relation on trains.) Give a satisfying assignment for the situation in which T_1 is on R_1 and T_2 is on R_3 in S_1.

11. Use the SAT algorithm to find a connected two-rail layout that causes deadlock for two trains. (A layout is connected if it's possible, in one or more steps, to move from every rail to every other rail).

12. Show that there is no connected three-rail layout that causes deadlock for two trains.

13. **Extra Credit:** Our definition of liveness in formula 8 is weak, because there still might be a train that never gets to move. But requiring every train to be able to move on every step is too strong. How could you axiomatize the notion that every train eventually gets to move? Would it work to try to prove or disprove such a liveness condition using the methods of this assignment? Why or why not?

14. **Extra Credit:** It might seem like the second implication ($\rightarrow$) in formula 5 should really be a biconditional ($\leftrightarrow$). Explain in English what that would mean, and why it would cause problems in our axiomatization.

4 Tools

We are making two tools available to help you in this assignment. The first is a Java program for converting formulae in first-order logic into CNF sentences. The second is a fast SAT solver, written in C++.

4.1 Converter

The converter rewrites a `Sentence` in full first-order logic into a propositional sentence, given a finite "universe" of `Objects`. Neither the `Universe` nor the first-order `Sentence` is mutated by the converter. We encourage you to make universes out of `Strings` with unique names. We will be using the "unique names" hypothesis, which is that items with different names are not

equal. Therefore, the objects in your universe should return distinct names via their `toString()` methods. It is these names that will be used when translating the `Equals` relation between two terms.

This converter *does not* handle `Functions`, so the only terms in your FOL sentences should be constant terms and variables. Any symbol that looks like it might be a variable, but is not bound by any quantifier is taken to be a constant.

For more detailed information on the operation of the converter, see the javadocs.

4.1.1 The Parser

The converter comes with a parser that takes strings and converts them into internal parse trees of first-order logic. The grammar of the language is shown in figure 3. Here are some example sentences:

```
foo ^ (~bar v baz v biz)
(foo -> bar) -> baz
~(exists x p(x))
all x exists y (p(x) -> r(x,y))
p(f(x,y),x,y)
all x exists y all z exists w r(x,y,z,w)
~q v T v F ^ p
```

4.1.2 Obtaining and using the code

Figure 4 shows two example usages of the converter code. The propositional sentences resulting from the conversion will be *really* big, in general. Printing them out using the `toString` method seems to run out of memory frequently. If you want to print out the intermediate result into a file, use the `convertFile` routine, which prints more efficiently.

Figure 5 shows an example input file. Blank lines are allowable, but remember that axioms must be connected with ^ symbols.

4.2 zChaff

You ought to be able to use your SAT solver from part 1a to do this assignment. But if your solver is too slow, and/or you want to try much bigger problems, you might want to use zChaff. Instructions for its use and installation are on the assignment web page. We have binaries for Linux and Solaris, and C++ source for the brave.

4.3 Debugging Hints

Writing axioms is a lot like writing a program. You'll almost always do it wrong the first time, and so you need to invent and use debugging strategies. Here are some that we've found useful in developing this assignment:

- If a set of axioms is unsatisfiable and you don't think it should be, remove constraints until you get a satisfying assignment. Be sure that assignment is reasonable given the constraints you think you've written down. Then gradually add constraints back in.

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{Sentence}_1 \ \texttt{<->} \ \textit{Sentence} \mid \textit{Sentence}_1 \\
\textit{Sentence}_1 & \rightarrow & \textit{Sentence}_2 \ \texttt{->} \ \textit{Sentence} \mid \textit{Sentence}_2 \\
\textit{Sentence2} & \rightarrow & \textit{Sentence}_3 \ \texttt{v} \ \textit{Sentence} \mid \textit{Sentence}_3 \\
\textit{Sentence}_3 & \rightarrow & \textit{Sentence}_4 \ \texttt{\^{}} \ \textit{Sentence} \mid \textit{Sentence}_4 \\
\textit{Sentence}_4 & \rightarrow & \texttt{\~{}} \ \textit{Sentence} \mid \\
 & & \quad \texttt{exists} \ \ \textit{SymbolSentence} \mid \\
 & & \quad \texttt{all} \ \textit{Symbol} \ \textit{Sentence} \mid \\
 & & \quad \texttt{(} \ \textit{Sentence} \ \texttt{)} \ \mid \\
 & & \textit{Proposition} \\
\textit{Proposition} & \rightarrow & \textit{AtomicProposition} \mid \textit{CompoundProposition} \mid \ \texttt{T} \ \mid \ \texttt{F} \\
\textit{CompoundProposition} & \rightarrow & \textit{RelationSymbol}(\textit{TermList}) \\
\textit{TermList} & \rightarrow & \epsilon \mid \textit{Term} \ \ \texttt{,} \ \ \textit{TermList} \\
\textit{Term} & \rightarrow & \textit{ConstantSymbol} \mid \textit{Variable} \mid \textit{FunctionSymbol} \ \ \texttt{(} \ \ \textit{TermList} \ \ \texttt{)}
\end{array}
$$

The non-terminals *PropositionVariable*, *RelationSymbol*, *ConstantSymbol*, *Variable*, and *FunctionSymbol* can all be groups of alphanumeric characters; they are not differentiated at the lexical level. We assume that any symbol not bound is a constant.
Note that the relation symbol `Equals` is syntactically the same as any other relation symbol, but it is treated specially by the converter.

Figure 3: Grammar for our FOL parser.

```
// Make the universe
Set u = new HashSet();
u.add("A");
u.add("B");
u.add("C");

// Convert an input file, written in FOL, to an output file,
// written in PL.
techniques.FOL.PLConverter.convertFile("input.txt", "output.txt", u)

// Read in an input file, convert it, and pass it to DPLL.
// Finally, print out a satisfying assignment
Sentence folS = techniques.FOL.parser.Parser.parseFile
                     ("/Users/lpk/Desktop/solution3.txt");
Sentence propS = techniques.FOL.PLConverter.convert(folS, u);
techniques.PL.Sentence cnfS =
    techniques.FOL.PLConverter.CNFconvert(propS);
techniques.PL.Interpretation answer =
    techniques.DPLL.solve((techniques.PL.Conjunction) cnfS);
System.out.println(answer);
```

Figure 4: Example usage of the converter

```
(all x all y all z r(x,y) ^ r(y,z) -> r(x,z))  ^

(all x all y r(x,y) -> r(y,x)) ^

r(A,B) ^
r(B,C) ^
~r(C,D)
```

Figure 5: Example input file

- Use plenty of parentheses. Be sure put one set of parentheses around each axiom in your input file.
- Be sure that you connect your axioms with a ^ symbol.
- Try a simpler version of what you're trying to do.
- Look at the CNF formula that is getting generated (sometimes this is helpful; sometimes it's just baffling).
- Always check to be sure that the assignments you're getting make sense.