First-Order Logic as a Lightweight Software Specification Language

The ETPTP Toolkit

Michel Rijnders
Master's Thesis
August 31, 2008

Master Software Engineering
University of Amsterdam

Host organisation:
Centrum voor Wiskunde en Informatica
Dutch National Research Institute for Mathematics and Computer Science

Thesis and Internship Supervisor:
Prof. Dr. D.J.N. van Eijck

Contents

Abstract

We investigate the use of first-order for lightweight software specification. We show that is possible to use existing tools to provide automatic analysis of specifications written in first-order logic plus (reflexive) transitive closure. The specifications are converted to pure first-order logic before they are passed to the model finder. This specification method is mostly model finder independent and it has been applied successfully to the development of several example specifications.

Chapter 1

Introduction

Formalization works as "an early-warning system" when things are getting contorted.

E.W. Dijkstra

1.1 Background and Context

Abstractions are fundamental to the design of software: good abstractions ease development and increase flexibility, whereas flawed abstractions impede development, cause defects, and hinder modification. Regrettably, abstractions are hard to get right, but formal methods, and especially formal specification, make better abstractions possible [12].

Formal methods are mathematically based techniques for the specification and verification of computing systems. Specification is the process of describing a system, its environment, or both. Verification is the process of establishing that a system has certain properties.

Specification is done using a specification language: a formal system which consists of syntax, semantics, and proof rules. Specification languages can be roughly classified as property-oriented or model-oriented [4]. Property-oriented languages, such as CafeOBJ [19] and CASL [3], emphasise properties of entities and functions, whereas model-oriented languages, such as VDM [17] and Z [20], emphasise mathematical values such as sets, maps, and sequences and functions over these values. The main benefit of the specification process is intangible: gaining a deeper understanding of the system in question [9], but there is a tangible byproduct, the written specification, which can be used in verification.

The two main methods for verification are deductive verification and model checking. Deductive verification, which is also know as inferential verification, is human-directed. Axioms and proof rules are used to prove that the system in question has certain properties. It is a time-consuming process which requires considerable expertise, therefore it is rarely used and it is applied mainly to highly sensitive systems such as security protocols [8]. Model checking, in contrast, aims to be fully automatic, and for that reason it is restricted to the verification of finite state machines. Properties, typically expressed in temporal logic, are verified by examining a model of the system in question. The main problem in model checking is the potential combinatorial explosion of the state space. This problem occurs in systems with many different interacting components or in systems that have data structures that contain many different values.

Full specification and verification are not economically viable for everyday software development, therefore lightweight formal methods [16] focus on the selective application of formal methods. Everyday software development can benefit most from analysis: analysis can expose specification and design errors when they are still inexpensive to fix. Lightweight formal methods aim to make analysis economically feasible by reducing the cost of specification and by automating the analysis itself. The cost of specification is reduced, first, by emphasising the tractability of specification languages; second, by encouraging partial analysis. The analysis is automated through model finding. Given a specification model finding looks for a model of the specification. On the surface model finding resembles model checking, but actually they are each other's inverse: model checking starts with a model and checks if the model satisfies a formula, while model finding starts with a formula and finds a model that satisfies the formula.

Lightweight formal methods emphasise the tractability of specification languages and thus favour small languages with simple semantics. This is strikingly different from conventional formal methods which view specification languages as general mathematical notations and accordingly emphasise expressiveness. First-order logic is the smallest, possibly still useful specification language we can think of. The goal of this project is to investigate the use of first-order logic for lightweight software specification.

1.2 Overview

In chapter 2 we investigate if it possible to express (reflexive) transitive closure in first-order logic first, because common connectivity and reachability properties cannot be expressed without (reflexive) transitive closure. Every example specifications in this report uses (reflexive) transitive closure. One of the specifications, for example, models the structural properties of a file system and uses the reflexive transitive closure of the “contents” relation of the root directory to express that the file system is connected.

As it turns out (reflexive) transitive closure can be expressed in first-order logic, at least for finite domains. This means it is possible, in theory, to use first-order logic for lightweight software specification, but *is it practical*? We noted above, for example, that automatic analysis is a key component of lightweight software specification. Can we use existing tools to provide automatic analysis of specifications written in first-order logic? This is possible as well. We have developed a toolkit that consists of an “off-the-shelf” model finder, a compiler which translates first-order logic plus (reflexive) transitive closure to first-order logic, and a visualisation tool. Chapter 3 introduces this toolkit. We assess the usability of the toolkit by applying it to the development of several example specifications. We also compare it with Alloy [14, 15], a state-of-the-art lightweight specification tool. Chapter 4 discusses the example specifications, chapter 5 contains the comparison with Alloy. Lastly, chapter 6 summarises our main findings, discusses the implications of those findings and outlines some ideas on future work.

Chapter 2

Expressing Transitive Closure in First-Order Logic

2.1 Preliminaries

Let R be a binary relation on some domain D.

Definition 2.1. The transitive closure R^+ of R is the smallest relation on D such that

(i) $R \subseteq R^+$, i.e., $\forall x, y.\ R(x,y) \rightarrow R^+(x,y)$

(ii) R^+ is transitive, i.e, $\forall x, y, z.\ R^+(x,y) \wedge R^+(y,z) \rightarrow R^+(x,z)$

Definition 2.2. The reflexive transitive closure R^* of R is the smallest relation on D such that

(i) $R \subseteq R^*$

(ii) R^* is transitive

(iii) R^* is reflexive, i.e., $\forall x.\ R^*(x,x)$

2.2 Transitive Closure in General

Transitive closure in general cannot be expressed in first-order logic. We give a proof by contradiction using the compactness theorem.

Compactness Theorem. *A set of sentences of first-order logic is satisfiable (has a model) iff every finite subset is satisfiable.*

Note that "sentence" here means "a formula without free variables".

Let L be a first-order language whose parameters include two binary predicate symbols R and T_R, and two constant symbols a and b. Let ϕ be a sentence that says that T_R is the transitive closure of R. Assume ϕ is satisfiable.

We define an infinite set Σ of sentences such that $\Sigma = \{\sigma_n \mid n \in \mathbb{N}\}$ where σ_0 is $\neg R(a,b)$ and where σ_n for any $n \geq 1$ is of the following form:

$$\forall x_1, \ldots, x_n.\ \neg(R(a,x_1)\ \wedge\ R(x_1,x_2)\ \wedge\ \ldots\ \wedge\ R(x_{n-1},x_n)\ \wedge\ R(x_n,b))$$

The sentence σ_n says that there is no R-path of length $n+1$ from a to b.

Let Γ be the infinite set of sentences such that $\Gamma = \{\phi, T_R(a,b)\} \cup \Sigma$. Every finite subset Γ' of Γ is satisfiable: a structure in which a R-path from a to b exists and in which the length l of every R-path from a to b is such that $l > n+1$ for every $\sigma_n \in \Gamma'$ is a model of Γ'.

Since every subset of Γ is satisfiable the compactness theorem gives us that Γ is satisfiable, but this leads to a contradiction: given $\{\phi, T_R(a,b)\} \subset \Gamma$ there is a R-path from a to b, but $\sigma_n \in \Gamma$ for every n, i.e., there is no R-path from a to b. $\square$

2.3 Transitive Closure on Finite Domains

We have just seen that transitive closure in general cannot be expressed in first-order logic, but the proof given is only valid for infinite domains. Model finding is restricted to finite domains [13], thus for our purposes it is enough if transitive closure on finite domains can be expressed in first-order logic. In this section we will see that this is possible.

2.3.1 Rewriting Transitive Closure

An naïve solution to the problem of expressing transitive closure on finite domains is to rewrite the transitive closure; given a domain of size n the following formula expresses the transitive closure of a relation R on that domain:

$$\forall x, y.\ R^+(x,y)\ \leftrightarrow\ R(x,y)\ \vee\ \sigma_1\ \vee\ \ldots\ \vee\ \sigma_{n-2}$$

where each sub-formula σ_n is of the following form:

$$\exists z_1, \ldots, z_n.\ R(x,z_1)\ \wedge\ R(z_1,z_2)\ \wedge\ \ldots\ \wedge\ R(z_{n-1},z_n)\ \wedge\ R(z_n,y)$$

The sub-formula σ_n says that there is a R-path of length $n+1$ from x to y.

Though easy to understand this solution suffers some problems. Either the domain size has to be known beforehand, or an extra sub-formula has to be added for every increase in domain size. But, more importantly, it introduces loads of auxiliary variables which slows down model finding, as we will see in chapter 5.

2.3.2 Axiomatising Transitive Closure

Koen Claessen [6] shows that is possible to axiomatise transitive closure on finite domains. This method of expressing transitive closure on finite domains does, of course, work for domains of arbitrary size, in contrast to the rewriting method discussed above. Claessen's axiomatisation relies upon the introduction of auxiliary symbols. Let's look at the axioms.

Axioms

Given a binary relation R on some domain the axioms define the symbol T_R as the reflexive transitive closure of R.

The first three axioms echo definition 2.2 above. They express that T_R is reflexive (I1), that T_R includes R (I2), and that T_R is transitive (I3).

$$\begin{array}{lc} \text{(I1)} & \forall x.\ T_R(x,x) \\ \text{(I2)} & \forall x,y.\ R(x,y) \rightarrow T_R(x,y) \\ \text{(I3)} & \forall x,y,z.\ T_R(x,y) \wedge T_R(y,z) \rightarrow T_R(x,z) \end{array}$$

But this is not enough: T_R must be the *smallest* relation satisfying these axioms. We must require there to be a R-path from a to b if $T_R(a,b)$.

The first two axioms we add require there to be a "next step" c from a towards b such that $R(a,c)$ and $T_R(c,b)$ if $T_R(a,b)$. The axioms introduce a new function symbol s_R; the term $s_R(a,b) = c$ indicates that c is a next step from a towards b:

$$\begin{array}{ll} \text{(E1)} & \forall x,y.\ T_R(x,y) \wedge x \neq y \rightarrow R(x, s_R(x,y)) \\ \text{(E2)} & \forall x,y.\ T_R(x,y) \wedge x \neq y \rightarrow T_R(s_R(x,y),y) \end{array}$$

If $x = y$ then there is no need to constrain $s_R(x,y)$ because (I1) already gives $T_R(x,y)$.

The axioms (E1) and (E2) do not constrain T_R enough. Figure 2.1 shows a counter example. Straight arrows denote R and dashed arrows denote T_R.

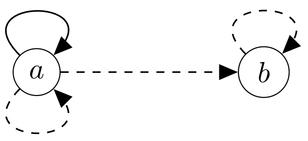

Figure 2.1: Counter example to (I1),(I2),(I3),(E1), and (E2).

The problem in the counter example is that $s_R(a,b) = a$. This combined with $R(a,a)$ makes the axioms (E1) and (E2) true for $\langle a,b \rangle$.

We could try to fix this by adding axiom (E3').

$$\text{(E3')} \quad \forall x,y.\ T_R(x,y) \wedge x \neq y \rightarrow s_R(x,y) \neq x$$

Axiom (E3') does not fix the problem though. Figure 2.2 gives a new counter example where $s_R(a,c) = b$ and $s_R(b,c) = a$. This makes the axioms (E1) and (E2) true for $\langle a,c \rangle$ and $\langle b,c \rangle$.

To solve this problem we observe the following: if we have $T_R(a,b)$ then we want $s_R(a,b)$ to be closer to b than a. We introduce another symbol: C_R, the term $C_R(a,b,c)$ indicates that b is closer to c than a. For each element e of the domain we want $C_R(x,y,e)$ to be a strict partial ordering (irreflexive and transitive):

$$\begin{array}{ll} \text{(C1)} & \forall x,y.\ \neg C(x,x,y) \\ \text{(C2)} & \forall x,y,z,v.\ C(x,y,v) \wedge C(y,z,v) \rightarrow C(x,z,y) \end{array}$$

The final axiom connects the symbol C_R with the other symbols:

$$\text{(E3)} \quad \forall x,y.\ T_R(x,y) \wedge x \neq y \rightarrow C(x, s_R(x,y), y)$$

The axiomatisation just presented only works for binary predicates, but it can easily be adapted for predicates with more arguments: the extra

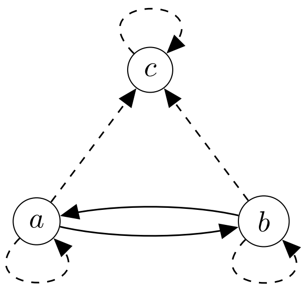

Figure 2.2: Counter example to (I1),(I2),(I3),(E1),(E2), and (E3').

argument positions that are not involved in the transitive closure must be added to the auxiliary symbols and all the axioms must get a number of extra universally quantified variables.

Alternative Axioms

Jan van Eijck (personal communications, June, 2008) came up with an alternative axiomatisation of transitive closure on finite domains as a by-product of trying to understand Claessen's axioms.

Let R be a binary relation on some domain. We use R^* for the reflexive transitive closure of R and R^+ for the transitive closure of R. We use I for the identity relation. For $n \in \mathbb{N}$ we define R^n by induction:

$$R^0 = I$$
$$R^{n+1} = R \circ R^n$$

Where $\circ$ denotes relational composition.

Assume the domain is finite, then $R^* = \bigcup_{n \geq 0} R^n$ and $R^+ = \bigcup_{n>0} R^n$.

We introduce a ternary relation symbol C whose intended interpretation is:

$$\lambda xyz.\exists n, m \in \mathbb{N}\ (n > 0 \wedge xR^n yR^m z \wedge \forall k \in \mathbb{N}(k < n + m \rightarrow \neg xR^k z)\ ).$$

The term $Cxyz$ indicates that y is at some non-zero distance from x on a shortest R-path from x to z.

Given that the domain is finite $\lambda xy.Cxyy$ expresses $R^+ - I$, accordingly we can define reflexive transitive closure using a binary relation symbol T_R as follows:

$$\text{(DEF)} \quad \forall x, y.\ T_R xy \rightarrow x = y \vee Cxyy$$

Clearly, $\lambda xy.Cxyu$ is irreflexive and transitive for any u.

$$\text{(C1)} \quad \forall x, u.\ \neg Cxxu$$
$$\text{(C2)} \quad \forall x, y, z, v.\ Cxyu \wedge Cyzu \rightarrow Cxzu$$

Axiom (C3) expresses that $R^+ - I$ is almost transitive. Axiom (C4) expresses that $R - I \subseteq R^+ - I$.

$$\text{(C3)} \quad \forall x, y, z.\ Cxyy \wedge Cyzz \wedge x \neq z \rightarrow Cxzz$$
$$\text{(C4)} \quad \forall x, y.\ Rxy \wedge x \neq y \rightarrow Cxyy$$

Axiom (C5) expresses that if $(x, y) \in (R^+ - I)$ then it is possible to make a first R-step on a shortest R-path from x to y.

$$\text{(C5)} \quad \forall x, y.\ Cxyy \rightarrow \exists z.\ Rxz \wedge Cxzy$$

Finally, axiom (C6) expresses that if y is somewhere along a shortest R-path from x to z and $y \neq z$, then $(y, z) \in (R^+ - I)$.

$$\text{(C6)} \quad \forall x, y, z.\ Cxyz \wedge y \neq z \rightarrow Cyzz$$

Chapter 3

The ETPTP Toolkit

The ETPTP toolkit provides automatic analysis for specifications written in ETPTP: a notation for first-order logic plus (reflexive) transitive closure based on TPTP [21]. TPTP is the format used by the TPTP Problem Library [22]: a standard set of test problems for automated theorem proving systems. The toolkit consists of the following components:

- a compiler that uses one of the axiomatisations of (reflexive) transitive closure described in the previous chapter to translate ETPTP to TPTP
- an "off-the-shelf" model finder that accepts TPTP input
- a visualisation tool that translates raw model finder output to the Dot format

The Dot format is part of the graph visualisation system Graphviz [11]. Figure 3.1 shows a diagram of the ETPTP work flow.

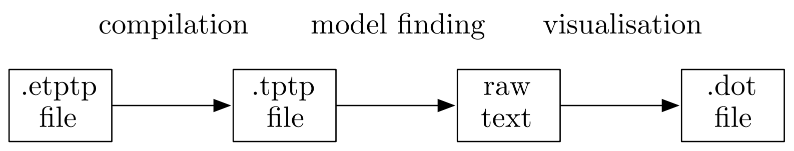

Figure 3.1: ETPTP work flow

During our research we have mainly used the Paradox model finder [7]. We selected Paradox because it has won the SAT/Models division of the CADE ATP System Competition [23], the world championship for automated first-order theorem provers, every year since 2003. The toolkit is mostly model finder independent though, and we have used the Darwin theorem prover [2] as well.

The toolkit is command line based. The compiler is called `etptp-e`, and the visualisation tool is called `model2dot`. Both are written in Haskell; appendix B lists the source code. Usage of the toolkit, with Paradox as its model finder, typically looks as follows:

```
$ etptp-e < example.etptp > example.tptp && \
> paradox --model example.tptp | \
> model2dot > example-model.dot
```

We will discuss the compiler and the visualisation tool in detail below, but let's look at the ETPTP format first.

3.1 The ETPTP Format

An ETPTP specification is basically a list of statements. ETPTP has four types of statements: include directives, comments, annotated formulae, and declarations. We will introduce these types with the help of a simple example specification. The example models structural properties of a file system. The next chapter explains the full details of this example. Note that the line numbers are not part of the specification. They are added for ease of reference.

```
 1  include: 'minimum-domain-size-5.tptp'
 2
 3  classes: file,dir
 4  axiom: (parent(X,Y) & parent(X,Z)) => Y = Z
 5  axiom: contents(X,Y) => dir(X)
 6  axiom: contents(X,Y) <=> parent(Y,X)
 7  axiom: dir(root) & -parent(root,X)
 8  % the file system is connected
 9  axiom: contents*(root,X)
10
11  % is the file system acyclic?
12  conjecture:
13    ?[X,Y] : (contents(X,Y) & contents+(Y,X))
```

3.1.1 Include Directives

The example starts with an include directive. ETPTP's include directive works like C's `#include` directive: the directive is replaced with the contents of the file. The file name must be enclosed in single quotes. The directive can span multiple lines.

3.1.2 Comments

ETPTP has single- and multi-line comments. Single-line comments start with the `%` character. Multi-line comments which start with the `/*` character sequence and end with the `*/` character sequence.

3.1.3 Annotated Formulae

There are two kinds of formulae: axioms and conjectures. Lines 4 to 9 show examples of axioms; lines 12 and 13 show an example of a conjecture. Line 12 and 13 also show that formulae, like include directives, can span multiple lines. As usual axioms are accepted without proof, whereas conjectures are to be proved.

Formula Syntax

Variable, function, and predicate names start with a letter. The rest of the name can consist of letters, digits, and underscores. All letters in variable names must be upper case, whereas all letters in function and predicate names must be lower case. Table 3.1 lists the syntax for connectives, quantifiers, and equations.

first-order logic	ETPTP
$\neg P$	`-p`
$P \wedge Q$	`p & q`
$P \vee Q \vee R$	`p | q | r`
$P \rightarrow Q$	`p => q`
$P \leftrightarrow Q$	`p <=> q`
$\forall x, y.\ R(x, y)$	`![X,Y] : r(X,Y)`
$\exists x.\ S(x)$	`?[X] : s(X)`
$x = y$	`X = Y`
$x \neq y$	`X != Y`

Table 3.1: ETPTP syntax for connectives, quantifiers, and equations

The notation for (reflexive) transitive closure is like the one used in chapter 2: the `+` character denotes transitive closure and the `*` character denotes reflexive transitive closure. Line 9 of the example uses reflexive transitive closure, line 13 uses transitive closure.

3.1.4 Declarations

There are two kinds of declarations: `classes`-declarations and `singletons`-declarations. Line 3 of the example shows a `classes`-declaration; `classes`-declarations are used to declare a number of mutually exclusive unary predicates, thus partitioning the domain, `singletons`-declarations are used to declare a number of unique constants.

3.2 Compiler

The `etptp-e` compiler is a source-to-source compiler: it translates first-order logic plus (reflexive) transitive closure in ETPTP format to first-order logic in TPTP format.

3.2.1 Front-End

The front-end of the compiler is essentially a combinator parser based on the Parsec library [18]. It combines lexical, syntactic, and semantic analysis. An ETPTP specification is a list of statements. Accordingly, the parser returns a list of instances of the `Statement` data type. The `Statement` data type is defined as follows:

```
data Statement = Include FilePath
               | AnnotatedFormula { name    :: Name
                                  , role    :: Role
                                  , formula :: Formula
                                  }
```

Recall that ETPTP has four types of statements: comments, include directives, formulae, and declarations. The data type has constructors for only two of these four types: there is no constructor for comments and there is no constructor for declarations. Comments are treated as white space and are discarded during parsing. Declarations are converted to formulae. Consider, for example, the `classes`-declaration from the example above:

```
classes: file,dir
```

This declares two mutually exclusive unary predicates `file` and `dir`. For this declaration the parser returns the same value as it would for the following formula:

```
axiom: (file(X) | dir(X))
       & (file(X) => -dir(X))
       & (dir(X) => -file(X))
```

That is, a `classes`-declaration that declares the predicates $p_1, \ldots, p_n$ is replaced with a formula of the form:

$$\forall x.\ (p_1(x) \lor \ldots \lor p_n(x)) \land \sigma_1 \land \ldots \land \sigma_n$$

where each sub-formula σ_m has the following form:

$$p_m(x) \rightarrow \neg(p_1(x) \lor \ldots \lor p_{m-1}(x) \lor p_{m+1}(x) \lor \ldots \lor p_n(x))$$

The sub-formula σ_m expresses that if $p_m(x)$ is true, then $p_k(x)$ is false for all $k \neq m$. The parser treats `singletons`-declarations similarly. A declaration of the constants $c_1, \ldots, c_n$ is replaced with a conjunction of clauses of the form $c_x \neq c_y$ for every pair of constants where $x \neq y$.

ETPTP formulae are implicitly universally quantified. TPTP, in contrast, does not allow free variables, therefore all free variables are bound during parsing. After replacing the declarations and after binding all free variables the ETPTP statements are handed to the back-end of the compiler.

3.2.2 Back-End

The back-end of the compiler converts ETPTP statements to TPTP statements. This mainly involves replacing (reflexive) transitive closure symbols and adding axioms. Reflexive transitive closure symbols, such as `contents*` on line 9 of our example, are replaced with a symbol where the asterisk has been changed to `_RTC`. Transitive closure symbols, such as `contents+` on line 13 of our example, are replaced with a symbol where the plus has been changed to `_TC`. Axioms are added for each replaced symbol; in our example both symbols use the binary predicate `contents`, thus the following set of axioms is added:

```
axiom: contents_TC(X,Y)
         <=> ?[Z] : (contents(X,Z) & contents_RTC(Z,Y))

axiom: contents_RTC(X,X)

axiom: contents(X,Y) => contents_RTC(X,Y)

axiom: (contents_RTC(X,Y) & contents_RTC(Y,Z))
         => contents_RTC(X,Z)

axiom:(contents_RTC(X,Y) & X != Y)
```

```
        => contents(X,contents_S(X,Y))

axiom: (contents_RTC(X,Y) & X != Y)
         => contents_RTC(contents_S(X,Y),Y)

axiom: (contents_RTC(X,Y) & X != Y)
         => contents_C(X,contents_S(X,Y),Y)

axiom: -contents_C(X,X,V)

axiom: (contents_C(X,Y,V) & contents_C(Y,Z,V))
         => contents_C(X,Z,V)
```

Note that although they are presented in ETPTP format here the back-end, obviously, uses an abstract syntax representation.

Finally, the two remaining types of statements, include directives and annotated formulae, are converted to their TPTP counterparts. This done by a pretty-printer to keep the object code readable.

3.3 Visualisation Tool

The visualisation tool is based on Graphviz [11]. It converts raw model finder output to a graph using the Dot format. Figure 3.2 shows a model of the example specification we have been using throughout this chapter.

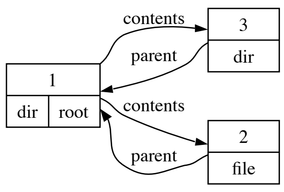

Figure 3.2: visualisation example

The example only has binary predicates and we want to explain the visualisation of predicates of greater arity as well, therefore figure 3.3 is added. It shows a model of a specification we will see in the next chapter. That specification models operations on a file system.

The numbered nodes of the graph represent the objects in the domain of the model. The labels on the nodes, such as “dir” and “fsys”, represent

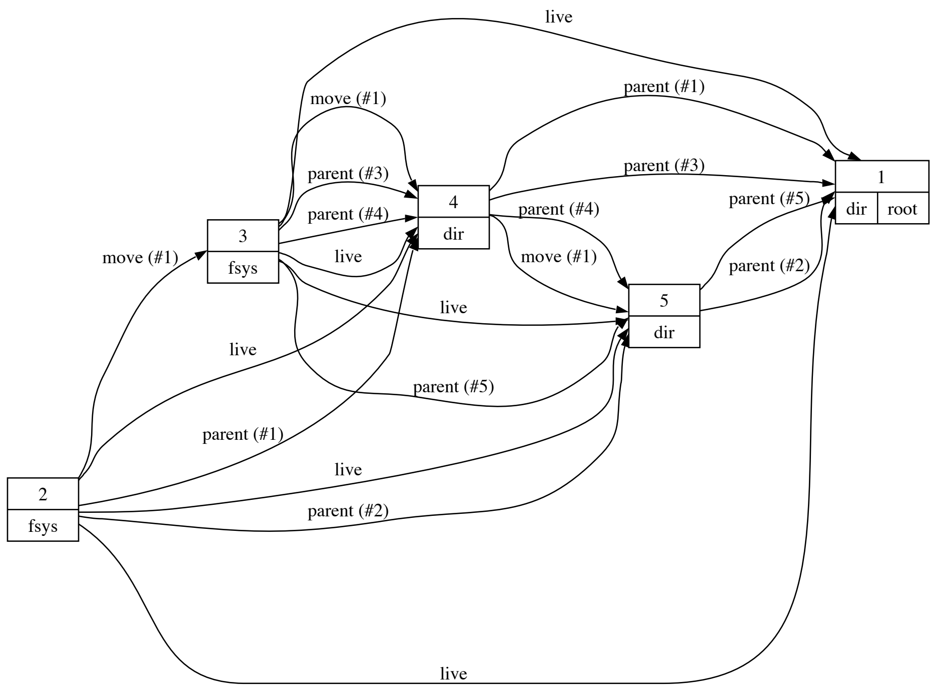

Figure 3.3: extended visualisation example

nullary functions and unary predicates. Functions and predicates with a greater arity, such as, "live" and "move", are represented by edges, where n-ary functions are treated like $n+1$-ary predicates. Unnumbered edges represent binary predicates, whereas numbered edges represent predicates whose arity is greater than two. For example, in figure 3.3 "live" is a binary predicate, while "move" is a quaternary predicate. Six tuples satisfy "live": $\langle 2,1\rangle$,$\langle 2,4\rangle$,$\langle 2,5\rangle$,$\langle 3,1\rangle$,$\langle 3,4\rangle$, and $\langle 3,5\rangle$, but only one tuple satisfies "move": $\langle 1,2,4,3\rangle$.

The conversion from raw model finder output to Dot format consist of the following steps:

1. The input from the model finder is preprocessed. This is the only part of our toolkit that is model finder dependent. The input is filtered and rewritten to an intermediate format. After preprocessing the input consist of strings representing equations and equivalences, such as `"root=1"` and `"parent(2,1)<=>$true"`.

2. The preprocessed input is parsed and converted to instances of a data type that represents equivalences. Equations are treated as equiva-

lences whose right-hand side is `$true`, thus `"root=1"` is treated like `"root(1)<=>$true"`.

3. The equivalences whose right-hand side is `$true` after substitution are converted to nodes and edges of a graph.

4. The resulting graph is printed using the Dot format.

The next chapter contains more examples of the use of the visualisation tool and the rest of the toolkit.

Chapter 4

Example ETPTP Specifications

We will now describe our example specifications: two introductory examples and two more advanced examples. We discuss the introductory examples almost in their entirety, whereas we only discuss some features of the advanced examples. Appendix A contains the complete listings of all four examples plus the listings of their Alloy counterparts.

4.1 Introductory Examples

The introductory examples are based on the first two examples of the online Alloy tutorial [1]. They both concern file systems. The first specification provides a static model of a file system, whereas the second specification provides a dynamic model of a file system. The use of these examples is twofold. First, they formed a feasibility study, second, they give a feel for what specification with the ETPTP toolkit is like.

4.1.1 Example 1: Structural Properties of a File System

We've already seen this example in the previous chapter. It models several structural properties of a file system. We will now discuss all its details. Everything in a file system is either a file or a directory. The first line uses a `classes`-declaration to express this.

```
classes: file,dir
```

Directories have contents, whereas files do not, thus if something has contents then it must be a directory.

```
axiom: contents(X,Y) => dir(X)
```

Where `contents(a,b)` means that `b` is part of `a`'s contents. A directory is the parent of its contents: the `parent` relation and the `contents` relation are each other's inverse.

```
axiom: contents(X,Y) <=> parent(Y,X)
```

We are modelling a very simple file system: it does not have symbolic links, accordingly, every object can have at most one parent.

```
axiom: (parent(X,Y) & parent(X,Z)) => Y = Z
```

The root directory, however, does not have a parent.

```
axiom: dir(root) & -parent(root,X)
```

Finally we want to express that the file system is connected: every object is reachable from the root directory, i.e., every object is in the reflexive transitive closure of the `contents`-relation of the root directory.

```
axiom: contents*(root,X)
```

Next we use the specification to find a model.

```
$ etptp-e < example1.etptp > example1.tptp && \
> paradox --model example1.tptp
```

Executing the preceding command gives the following output:

```
Paradox, version 2.3, 2007-11-01.
+++ PROBLEM: example1.tptp
Reading 'example1.tptp' ... OK
+++ SOLVING: example1.tptp
domain size 1
+++ BEGIN MODEL
% domain size is 1

contents(!1,!1) <=> $false

contents_C(!1,!1,!1) <=> $false

contents_RTC(!1,!1) <=> $true
```

```
contents_TC(!1,!1) <=> $false

contents_S(!1,!1) = !1

dir(!1) <=> $true

file(!1) <=> $false

parent(!1,!1) <=> $false

root = !1
+++ END MODEL
+++ RESULT: Satisfiable
```

Note the following:

- A model is found, albeit a very simple one: it consist of a single directory that is the root directory of the file system.
- Several auxiliary symbols have been introduced because of the use of the (reflexive) transitive closure of `contents`.

To get a more interesting model we add the following two lines to the specification.

```
singletons: obj1,obj2,obj3,obj4,obj5
axiom: ?[X] : file(X)
```

The `singletons`-declaration adds five unique objects to the domain. The extra axiom guarantees that one of the elements of the domain will be a file. Again we look for a model, but now we use the visualisation tool.

```
$ etptp-e < example1.etptp > example1.tptp && \
> paradox --model example1.tptp | \
> grep -v contents_ | grep -v obj | \
> model2dot > model.dot
```

We use grep to remove some predicates and functions, such as those introduced because of the use of the (reflexive) transitive closure of `contents`, from the model. Figure 4.1 shows the result.

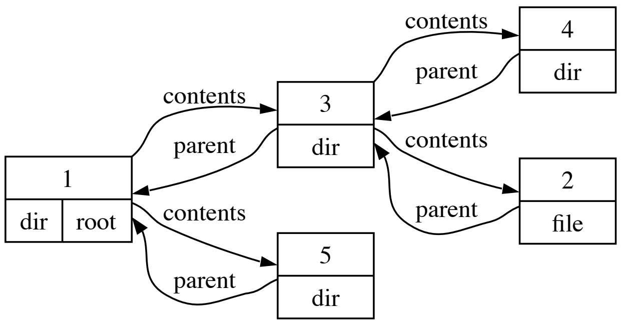

Figure 4.1: a model of the first example specification

4.1.2 Example 2: Operations on a File System

This second introductory example is an extension of the first one. It models operations on a file system. It shows how to model dynamic systems: the operations are modelled as transitions between two file system. The file systems represent the state before and the state after the operation. We need a predicate for file systems, `fsys`, and a predicate that relates file systems to the files and directories they contain, `live`.

```
classes: fsys,dir,file
axiom: live(X,Y) => (fsys(X) & -fsys(Y))
```

The domain now contains multiple file systems, but they are intended as different versions of the same file system and the operations we model do not change the root directory, therefore the root directory remains a constant. Every version of the file system contains the root directory.

```
axiom: dir(root) & (fsys(X) => live(X,root))
```

The other predicates, `contents` and `parent`, do change: they change from binary predicate to ternary predicate. The extra argument represents the version of the file system, for example `parent(a,b,c)` means “in version `a` of the file system, `c` is `b`’s parent. As before, the root directory does not have a parent. We also use the new version of `parent` to constrain `live`.

```
axiom: -parent(X,root,Y)
axiom: (live(X,Y) & Y != root) <=> ?[V]: parent(X,Y,V)
```

The changes to the `contents` predicate are similar to the changes to the `parent` predicate. We use the transitive closure of `contents` to express that the file system is acyclic, and `contents` and `parent` remain each other's inverse.

```
axiom: contents(X,Y,Z) => (fsys(X) & dir(Y) & -fsys(Z))
axiom: contents+(X,Y,Z) => Y != Z
axiom: contents(X,Y,Z) <=> parent(X,Z,Y)
```

We are now ready to model a file system operation: moving a file, `move(a, b,c,d)` means "version `b` of the file system is the result of moving object `c` to directory `d` in version `a` of the file system".

```
axiom:
  move(FS1,FS2,X,D) =>
    ( (fsys(FS1) & fsys(FS2) & -fsys(X) & dir(D))
    & (X != root & X != D)
    & (live(FS1,X) & live(FS1,D) & -parent(FS1,X,D) & ↩
    ↪ parent(FS2,X,D))
    & ( Y!= X => (parent(FS2,Y,E) <=> parent(FS1,Y,E)))
    )
```

The consequent of the implication above expresses the following constraints:

- the root directory cannot be moved
- an object cannot be moved to itself
- the object to be moved and the directory it is moved to must be part of the "starting" file system
- only the parent of the moved object changes

This example models two more operations. The removal of a single file or an empty directory from a file system (`remove`) and the recursive removal of a file or directory from a file system (`remove_all`). We will not discuss these operations because their specification is lot like the specification of `move` above.

4.2 Advanced Examples

We will now discuss the two more advanced examples. We will only discuss the features that distinguish them from the introductory examples. The first of these two specifications models the solution to a river crossing puzzle. It is based on the last example of the Alloy tutorial. The second specification models a typical software design problem [24]: leader election in a ring.

4.2.1 Example 3: River Crossing Puzzle

This specification solves the well-known river crossing puzzle of a farmer who wants to transport a fox, a chicken, and a bag of grain across a river with a boat that can only hold one item in addition to the farmer. The fox cannot be left alone with the chicken and the chicken cannot be left alone with the grain.

The specification introduces the farmer, fox, chicken, and grain with a `singletons`-declaration. The axiom following the declaration groups these objects, thus introducing a notational shortcut.

```
singletons: farmer,fox,chicken,grain
axiom: object(X) <=>
  (X = farmer | X = fox | X = chicken | X = grain)
```

The sides of the river are introduced similarly.

```
singletons: none,near,far
axiom: river_side(X) <=>
         (X = none | X = near | X = far)
```

The function `side(x,y)` returns the side of the river an object `y` is on in state `x`. The `none` object is used to simulate a partial function.

```
axiom: side(X,Y) = none <=> -(state(X) & object(Y))
```

A solution to the puzzle is found by connecting the initial state and the final state with the transitive closure of the `next_state` relation. The `next_state` predicate puts the necessary constraints on the transition from one state to the next state.

```
axiom: ?[X,Y]: ( init_state(X)
               & final_state(Y)
               & next_state+(X,Y)
               )
```

4.2.2 Example 4: Leader Election in a Ring

The last example gives a specification of a well-known distributed algorithm for finding the largest of a set of uniquely numbered processes arranged in a ring [5]. The algorithm works as follows:

1. each process initially generates a message with its own number and passes the message to its left

2. a process receiving a message compares the number in the message to its own number
    - if its own number is lower, the process passes the message (again to its left)
    - if its own number is equal, the process declares itself leader
    - if its own number is higher, the process it discards the message

This specification has one feature the other specifications do not have: a subset of its domain, the processes, must be a totally ordered set, therefore the following axioms are added:

```
axiom: proc_lte(X,Y) => (proc(X) & proc(Y))
axiom: (proc_lte(X,Y) & proc_lte(Y,Z)) => proc_lte(X,Z)
axiom: (proc_lte(X,Y) & proc_lte(Y,X)) => X = Y
axiom: (proc(X) & proc(Y)) => ( proc_lte(X,Y)
                              | proc_lte(Y,X))
```

`proc_lte(a,b)` means "the number of process `a` is less than or equal to the number of process `b`". Note that these axioms use the "standard" way of defining a total order, using $\leq$, but restrict the order to certain elements of the domain: those satisfying the `proc` predicate. We will return to this example and especially to its use of total orderings in the next chapter which compares the ETPTP toolkit and Paradox with Alloy.

Chapter 5

Comparisons

We will now compare the ETPTP toolkit and Paradox with Alloy. Ideally lightweight software specification has the immediacy of unit testing. We focus on two aspects that can have a significant impact on that immediacy: model finding performance and notational efficiency.

5.1 Model Finding Performance

We have carried out three performance tests. All test were run on a virtual machine running under VMware Fusion on a 2GHz Intel Core 2 Duo MacBook. The virtual machine ran Linux and was assigned 1GB of memory and one virtual processor. We used version 4.1.6. of Alloy and version 3.0 of Paradox. Paradox uses MiniSat [10] as its SAT engine. Alloy was configured to use MiniSat as its SAT engine too.

The tests are based on the example specifications from the previous chapter. The first test is based on the first example, the second test is based on the second example, and the third test is based on the fourth example.

The first test compares the different ways of expressing (reflexive) transitive closure. The example specification was translated to TPTP in three different ways: using Koen Claessen's axioms (KC), using Jan van Eijck's axioms (JvE), and using the rewriting method from section 2.3.1 (*rewritten*). For reference the test also includes Alloy. Table 5.1 gives the results. Dashes indicate timeouts (no result within five minutes). The results show that the rewriting method performed poorly, as expected. There is not much difference between the two axiomatisations.

The second test measures the time needed to prove the conjecture in the

	execution time (s)			
	Alloy	Paradox		
domain size		KC	JvE	rewritten
5	0.05	0.19	0.06	0.17
6	0.14	0.24	0.08	0.45
7	0.87	0.27	0.12	3.88
8	9.30	0.37	0.20	203.94
9	69.18	1.07	2.57	-
10	230.38	18.09	8.96	-
11	-	204.60	200.95	-
12	-	-	-	-

Table 5.1: Time needed to prove the conjecture in the first example specification.

second example specification for a domain consisting of two file systems and a given number of files and directories. Table 5.2 lists the results. The third test measures the time needed to find a model of the fourth example specification for a given number of processes. The results are in table 5.3. In the first test the performance of Paradox was better than the performance of Alloy. Here, we see the reverse. The difference on the third test is noticeable, but easy to explain: Alloy has built-in optimisations for specifications involving total orderings.

	execution time (s)	
files and directories	Alloy	Paradox
5	0.14	0.89
7	0.14	1.24
9	0.23	1.69
11	0.36	5.00
13	0.56	3.98
15	1.19	32.45

Table 5.2: Time needed to prove the conjecture in the second example specification.

	execution time (s)	
processes	Alloy	Paradox
3	0.08	0.55
4	0.06	0.68
5	0.09	0.96
6	0.12	5.58
7	0.19	-

Table 5.3: Time needed to find a model of the fourth example specification.

Our concern here is not which of the two model finders is the fastest though. Alloy is used as a reference point and we want to verify that Paradox's performance is at least acceptable. The test results confirm this, especially under Daniel Jackson's small scope hypothesis [15]. The small scope hypothesis claims that most software defects have small counterexamples.

5.2 Notational Efficiency

This section compares the notational efficiency of ETPTP and Alloy. Let's define what we mean by notational efficiency first. Suppose we have two semantically equivalent specifications written in two different specification languages L_1 and L_2. If the specification written in L_1 is smaller than the specification written in L_2, then we would call L_1 more efficient than L_2.

To assess the relative notational efficiency of Alloy and ETPTP we measured the size of example specification examples written in both languages. We first counted the number of lines filtering empty lines and comments. Table 5.4 lists the results. These result hardly show any difference between Alloy and ETPTP, but a review of the example specifications at least gives the impression that Alloy is somewhat more efficient than ETPTP. A count of the number of characters in the specifications, table 5.5, confirmed this. Again, the difference for the fourth example is noticeable, and again is has to do with the use of total orderings in the specification. The Alloy specification includes a library module for total orderings which we do not include in the count. The size of that library module is 21 lines or 813 characters.

Again we use Alloy as a reference point, but now for the verification of ETPTP's notational efficiency. The test results confirm that ETPTP's notational efficiency is at least acceptable.

	line count	
example	Alloy	ETPTP
1	8	7
2	30	34
3	31	32
4	30	37

Table 5.4: Size of the example specifications in lines.

	character count	
example	Alloy	ETPTP
1	324	243
2	838	1075
3	819	1073
4	726	1406

Table 5.5: Size of the example specification in characters.

Chapter 6

Conclusion

6.1 Main Findings

We have investigated the use of first-order logic for lightweight software specification. We have shown that it is possible to use existing tools to provide automatic analysis of specifications written in first-order logic plus (reflexive) transitive closure, where the specifications are converted to pure first-order logic before being passed to a model finder. The conversion to pure first-order logic is based on an axiomatisation of (reflexive) transitive closure on finite domains. This specification method is mostly model finder independent and it has been applied successfully to the development of several example specifications.

Our specification method has two advantages. First, it uses a standard logic; second, it is mostly model finder independent. This allows us to benefit from a large body of existing and future research on first-order logic, theorem proving, and model finding. In fact, our application of Koen Claessen's work is already an example of this. Our specification method also gives existing theorem provers and model finders a new application area.

During our research we mainly used Paradox as our model finder. We have compared the performance of Paradox with the performance of Alloy on the example specifications. Alloy's performance is better than Paradox's performance, but Paradox's performance is adequate for lightweight software specification.

We have also compared the sizes of the example specifications and their Alloy counterparts. The Alloy specifications are, on average, 25% shorter.

6.2 Future Work

We would like to further extend the `etptp-e` compiler, thus allowing us to remove boilerplate code from the example specifications. We are considering the following extensions:

- Orderings declarations: the fourth example specification basically contains the same set of axioms twice to make two of its sub-domains total orderings. We could extend `classes`-declarations and add those axioms automatically. The first line of the fourth example would then look something like `classes: state[TO],proc[TO]`. Of course, this would also allow for partial orderings.
- Notation for multiplicity constraints: this would introduce notational shortcuts for formulae such as:
  - `(p(X) & p(Y)) => X = Y`
  - `?[X,Y]: (X != Y & p(Z) => (Z = X | Z = Y))`
  - `p(X) => (X = a | X = b | X = c)`
- Sub-formula sharing: some definitions, like `remove` and `remove_all` in the second example, have a lot of sub-formulae in common. A mechanism to define and share these common sub-formula would an improvement.

Care should be taken tough that we do not stray too far away from pure first-order logic.

We can easily imagine all kinds of improvements to the visualisation tool. Some way to interactively manipulate the graph and thus being able, for example, to filter out certain relations would especially enhance the usability of ETPTP toolkit.

We also need to improve the error messages of the ETPTP parser.

Bibliography

[1] Alloy tutorial [online]. Available from: `http://alloy.mit.edu/alloy4/tutorial4/index.html`.

[2] Baumgartner, P., and Tinelli, C. Darwin: A Theorem Prover for the Model Evolution Calculus. *IJCAR Workshop on Empirically Successful First Order Reasoning* (2004).

[3] Bidoit, M., and Mosses, P. *CASL User Manual: Introduction to Using the Common Algebraic Specification Language.* Springer, 2004.

[4] Bjørner, D., and Henson, M., Eds. *Logics of Specification Languages.* Springer, 2008.

[5] Chang, E., and Roberts, R. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM 22*, 5 (1979), 281–283.

[6] Claessen, K. Expressing Transitive Closure for Finite Domains in Pure First-Order Logic. Tech. rep., Chalmers University of Technology. Forthcoming.

[7] Claessen, K., and Sörensson, N. New Techniques that Improve MACE-style Finite Model Finding. *CADE-19, Workshop W 4* (2003).

[8] Clarke, E., Grumberg, O., and Peled, D. *Model checking.* The MIT Press, 1999.

[9] Clarke, E. M., and Wing, J. M. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv. 28*, 4 (1996), 626–643.

[10] Eén, N., and Sörensson, N. An Extensible SAT-solver. *Lecture Notes In Computer Science* (2004), 502–518.

[11] Gansner, E. R., and North, S. C. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience 30*, 11 (2000), 1203–1233.

[12] Guttag, J., and Horning, J. Formal specification as a design tool. In *Proceedings of the Seventh ACM SIGPLAN-SIGACT Symposium on Principles of Programming.* (1980), ACM Press New York, NY, USA.

[13] Jackson, D. Automating first-order relational logic. *SIGSOFT Softw. Eng. Notes 25*, 6 (2000), 130–139.

[14] Jackson, D. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology 11*, 2 (2002), 256–290.

[15] Jackson, D. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[16] Jackson, D., and Wing, J. Lightweight Formal Methods. *Computer 29*, 4 (april 1996), 21–22.

[17] Jones, C. *Systematic software development using VDM.* Prentice Hall New York, 1990.

[18] Leijen, D., and Meijer, E. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science 41*, 1 (2001).

[19] Nakajima, S., and Futatsugi, K. An object-oriented modeling method for algebraic specifications in CafeOBJ. In *ICSE '97: Proceedings of the 19th international conference on Software engineering* (New York, NY, USA, 1997), ACM, pp. 34–44.

[20] Spivey, J. *The Z notation: a reference manual.* Prentice Hall, 1992.

[21] Sutcliffe, G., Schulz, S., Claessen, K., and Van Gelder, A. Using the TPTP Language for Writing Derivations and Finite Interpretations. *Journal of Functional Programming 8*, 04 (2006), 437–444.

[22] Sutcliffe, G., and Suttner, C. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning 21*, 2 (1998), 177–203.

[23] Sutcliffe, G., and Suttner, C. The State of CASC. *AI Communications 19*, 1 (2006), 35–48.

[24] Torlak, E., and Jackson, D. Kodkod: A relational model finder. *Tools and Algorithms for Construction and Analysis of Systems (TACAS) 39* (2007), 63.

Appendix A

Example Specifications Listings

A.1 Example 1: Structural Properties of a File System

ETPTP

```
classes: file,dir

axiom: (parent(X,Y) & parent(X,Z)) => Y = Z
axiom: contents(X,Y) => dir(X)
axiom: contents(X,Y) <=> parent(Y,X)
axiom: dir(root) & -parent(root,X)
% the file system is connected
axiom: contents*(root,X)

% is the file system acyclic?
conjecture: ?[X,Y]: (contents(X,Y) & contents+(Y,X))
```

Alloy

```
// A file system object in the file system
sig FSObject { parent: lone Dir }

// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }

// All file system objects are either files or directories
fact { File + Dir = FSObject }

// A directory is the parent of its contents
```

```
fact { all d: Dir, o: d.contents | o.parent = d }

// There exists a root
one sig Root extends Dir { } { no parent }

// File system is connected
fact { FSObject in Root.*contents }

// The contents path is acyclic
assert acyclic { no d: Dir | d in d.^contents }
```

A.2 Example 2: Operations on a File System

ETPTP

```
classes: fsys,dir,file

axiom: ?[X,Y,Z]: remove_all(X,Y,Z)

axiom: live(X,Y) => (fsys(X) & -fsys(Y))

axiom: dir(root) & (fsys(X) => live(X,root))
axiom: -parent(X,root,Y)
axiom: (live(X,Y) & Y != root) <=> ?[V]: parent(X,Y,V)

axiom: contents(X,Y,Z) => (fsys(X) & dir(Y) & -fsys(Z))
axiom: contents+(X,Y,Z) => Y != Z
axiom: contents(X,Y,Z) <=> parent(X,Z,Y)

axiom: move(FS1,FS2,X,D) =>
( (fsys(FS1) & fsys(FS2) & -fsys(X) & dir(D))
& (X != root & X != D)
& (live(FS1,X) & live(FS1,D))
& (-parent(FS1,X,D) & parent(FS2,X,D))
& (Y != X => (parent(FS2,Y,E) <=> parent(FS1,Y,E)))
)

axiom: remove(FS1,FS2,X) =>
( (fsys(FS1) & fsys(FS2) & -fsys(X))
& X != root
& (live(FS1,X) & -live(FS2,X))
& -?[Y]: parent(FS1,Y,X)
& ![Y,D]:
  (Y != X => (parent(FS2,Y,D) <=> parent(FS1,Y,D)))
)

axiom: remove_all(FS1,FS2,X) =>
( (fsys(FS1) & fsys(FS2) & -fsys(X))
& X != root
& (live(FS1,X) & -live(FS2,X))
& -?[D]: parent(FS2,X,D)
& ![Y,D]:
  (Y != X =>
    (parent(FS2,Y,D) <=> (parent(FS1,Y,D) & -contents+(FS1,X,Y))))
)

conjecture: move(FS1,FS2,X,D) => ![Y]: (live(FS1,Y) <=> live(FS2,Y))
```

Alloy

```
// File system objects
abstract sig FSObject { }
sig File, Dir extends FSObject { }

// A File System
sig FileSystem {
  live: set FSObject,
  root: Dir & live,
  parent: (live - root) ->one (Dir & live),
  contents: Dir -> FSObject
}{
  // live objects are reachable from the root
  live in root.*contents
  // parent is the inverse of contents
  parent = ~contents
}

// Move x to directory d
pred move [fs, fs': FileSystem, x: FSObject, d: Dir] {
  (x + d) in fs.live
  fs'.parent = fs.parent - x->(x.(fs.parent)) + x->d
}

// Delete the file or directory x
pred remove [fs, fs': FileSystem, x: FSObject] {
  x in (fs.live - fs.root)
  fs'.root = fs.root
  fs'.parent = fs.parent - x->(x.(fs.parent))
}

// Recursively delete the object x
pred removeAll [fs, fs': FileSystem, x: FSObject] {
  x in (fs.live - fs.root)
  fs'.root = fs.root
  let subtree = x.*(fs.contents) |
      fs'.parent = fs.parent - subtree->(subtree.(fs.parent))
}

// Moving doesn't add or delete any file system objects
moveOkay: check {
  all fs, fs': FileSystem, x: FSObject, d:Dir |
    move[fs, fs', x, d] => fs'.live = fs.live
} for 5
```

A.3 Example 3: River Crossing Puzzle

ETPTP

```
classes: state,object,river_side

singletons: farmer,fox,chicken,grain
axiom:
  object(X) <=> (X = farmer | X = fox | X = chicken | X = grain)

singletons: none,near,far
```

```
axiom: river_side(X) <=>  (X = none | X = near | X = far)

axiom: side(X,Y) = none | side(X,Y) = near | side(X,Y) = far
axiom: side(X,Y) = none <=> -(state(X) & object(Y))

axiom:
init_state(X) <=> (state(X) & ![Y]: (object(Y) => side(X,Y) = near))

axiom:
final_state(X) <=> (state(X) & ![Y]: (object(Y) => side(X,Y) = far))

axiom: eats(X,Y,Z) <=>
( state(X)
& side(X,Y) = side(X,Z)
& side(X,Y) != side(X,farmer)
& ((Y = fox & Z = chicken) | (Y = chicken & Z = grain))
)

axiom: next_state(X,Y) <=>
( state(X)
& state(Y)
& -?[Z,V] : eats(Y,Z,V)
& side(X,farmer) != side(Y,farmer)
& ![Z]: (( Z != farmer
           & side(X,Z) = side(X,farmer)
           & side(Y,Z) = side(Y,farmer)
         ) => (( Z = fox | Z = chicken | Z = grain)
               & ![V]: ((V != farmer & side(X,V) != side(Y,V))
                          => V = Z))))
axiom:
?[X,Y]: (init_state(X) & final_state(Y) & next_state+(X,Y))
```

Alloy

```
open util/ordering[State] as ord

abstract sig Object { eats: set Object }
one sig Farmer, Fox, Chicken, Grain extends Object {}

fact eating { eats = Fox->Chicken + Chicken->Grain }

sig State {
   near: set Object,
   far: set Object
}

fact initialState {
   let s0 = ord/first |
     s0.near = Object && no s0.far
}

pred crossRiver [from, from', to, to': set Object] {
  // either the Farmer takes no items
  ( from' = from - Farmer &&
    to' = to - to.eats + Farmer ) ||
  // or the Farmer takes one item
  (some item: from - Farmer {
    from' = from - Farmer - item
    to' = to - to.eats + Farmer + item
  })
}

fact stateTransition {
  all s: State, s': ord/next[s] {
    Farmer in s.near =>
      crossRiver[s.near, s'.near, s.far, s'.far] else
      crossRiver[s.far, s'.far, s.near, s'.near]
  }
}

pred solvePuzzle {
     ord/last.far = Object
}

run solvePuzzle for 8 State expect 1
```

A.4 Example 4: Leader Election in a Ring

ETPTP

```
classes: state,proc

axiom: ?[X]: init(X)
axiom: ?[X,Y]: lead(X,Y)

axiom: lead(X,Y) => (state(X) & proc(Y))
axiom: msg(X,Y,Z) => (state(X) & proc(Y) & proc(Z))
axiom: succ(X,Y) => (proc(X) & proc(Y))
axiom: init(X) => state(X)
axiom: next(X,Y) => (state(X) & state(Y))

% ring
axiom: proc(X) =>
( ?[Y] : (succ(X,Y) & ![Z] : (succ(X,Z) => Y = Z))
& ?[Y] : (succ(Y,X) & ![Z] : (succ(Y,Z) => X = Z))
& ![Y] : (proc(Y) => succ+(X,Y))
)

% initial state
axiom:init(X) =>
( state_fst(X)
& ![Y] : (proc(Y) => msg(X,Y,Y))
& ![Y,Z] : (msg(X,Y,Z) => Y = Z)
)

% state transition
axiom: (next(S1,S2) & succ(P1,P2)) =>
![P3]: (proc(P3)
        => (msg(S2,P2,P3) <=> ( msg(S1,P1,P3)
                                & proc_lte(P2,P3))))

axiom:
(state_lt(S1,S2) & -?[S3] : (state_lt(S1,S3) & state_lt(S3,S2)))
  => next(S1,S2)

% leader elected
```

```
axiom: lead(S,P) <=> (-init(S) & msg(S,P,P))

% total ordering of processes
axiom: proc_lte(X,Y) => (proc(X) & proc(Y))
axiom: (proc_lte(X,Y) & proc_lte(Y,Z)) => proc_lte(X,Z)
axiom: (proc_lte(X,Y) & proc_lte(Y,X)) => X = Y
axiom: (proc(X) & proc(Y)) => (proc_lte(X,Y) | proc_lte(Y,X))
axiom: proc_lt(X,Y) <=> (X != Y & proc_lte(X,Y))

% total ordering of states
axiom: state_lte(X,Y) => (state(X) & state(Y))
axiom: (state_lte(X,Y) & state_lte(Y,Z)) => state_lte(X,Z)
axiom: (state_lte(X,Y) & state_lte(Y,X)) => X = Y
axiom: (state(X) & state(Y)) => (state_lte(X,Y) | state_lte(Y,X))
axiom: state_lt(X,Y) <=> (X != Y & state_lte(X,Y))
axiom: state_fst(X) <=> ![Y] : (state(Y) => state_lte(X,Y))
```

Alloy

```
module ringlead

open util/ordering[Time] as TO
open util/ordering[Process] as PO

sig Time {
   elected: set Process
}

sig Process {
   succ: one Process,
   toSend: Process -> Time
}

fact ring { all p: Process | Process in p.^succ}

pred init [t: Time] { all p: Process | p.toSend.t = p }

pred step [t, t': Time, p: Process] {
   let from = p.toSend, to = p.succ.toSend
     | to.t' = from.t - p.succ.prevs
}

fact defineElected {
   no elected.first
   all t: Time-first
     | t.elected = {p: Process | p in p.toSend.t - p.toSend.(t.prev)↩
     ↪ }
}

fact traces {
  init [first]
  all t: Time-last |
    let t' = t.next |
      all p: Process |
        step [t, t', p]
}

pred show { some elected  }
run show for  5 Process, 6 Time
```

Appendix B

ETPTP Toolkit Source Code

B.1 Compiler

```
module MinE where

import ETPTP.FrontEnd
import ETPTP.BackEnd
import ETPTP.PrettyPrinter
import System.Exit
import System.IO
import Text.ParserCombinators.Parsec.Prim (parse)

main = do input <- hGetContents stdin
          parseResult <- return (parse parser "" input)

          case parseResult of
            Left error ->
              do hPutStrLn stderr ("PARSE ERROR " ++ show error)
                 exitFailure
            Right statements ->
              do hPutStrLn stdout (printer (toTPTP statements))
```

B.1.1 FrontEnd Module

```
module ETPTP.FrontEnd (parser) where

import Data.List (nub)
import ETPTP.AbstractSyntax hiding (formula)
import Text.ParserCombinators.Parsec
import Text.ParserCombinators.Parsec.Language
import qualified Text.ParserCombinators.Parsec.Token as P

---------------------------------------------------------------------------
-- Lexer
---------------------------------------------------------------------------

lexer = P.makeTokenParser
        (emptyDef
         { commentStart     = "/*"
         , commentEnd       = "*/"
         , commentLine      = "%"
         , identStart       = letter
         , identLetter      = alphaNum <|> char '_'
         , opStart          = oneOf "=<-&|!?"
         , opLetter         = oneOf ">="
         , reservedNames    = ["include"
                              ,"classes"
                              ,"singletons"
                              ,"axiom"
                              ,"conjecture"
                              ]
         , caseSensitive    = True
         })


colon      = P.colon      lexer
commaSep   = P.commaSep   lexer
lexeme     = P.lexeme     lexer
operator   = P.lexeme     lexer
parens     = P.lexeme     lexer
reserved   = P.reserved   lexer
reservedOp = P.reservedOp lexer
symbol     = P.symbol     lexer
whiteSpace = P.whiteSpace lexer

lowerSymbol :: Parser String
lowerSymbol = lexeme $ do c <- lower
                          cs <- many $ lower <|> digit <|> char '_'
                          return (c:cs)

upperSymbol :: Parser String
upperSymbol = lexeme $ do c <- upper
                          cs <- many $ upper <|> digit <|> char '_'
                          return (c:cs)

-----------------------------------------------------------------------
-- Parser
-----------------------------------------------------------------------

parser :: Parser [Statement]
parser = do whiteSpace
            ss <- many (include <|> declarations <|> annotatedFormula)
            eof
            return ss

-- Statements
include :: Parser Statement
include = let file = do char '\''
                        f <- manyTill anyChar (char '\'')
                        return f
          in
            do reserved "include"
               colon
               f <- lexeme file
               return $ Include f

declarations :: Parser Statement
declarations = classes <|> singletons

classes :: Parser Statement
classes   = do reserved "classes"
               colon
               ss <- commaSep lowerSymbol
               return $ AnnotatedFormula "classes" Axiom (formula ss)
  where
    formula :: [Symbol] -> Formula
    formula ss = f
      where
        f  = Forall [x] (And [f1,f2])
        f1 = Or (map (\s -> predicate s) ss)
        f2 = And (map (\(s1,s2) -> notBoth s1 s2) (combis ss))
        x  = Variable "X"

        predicate s = Atom (Function s [x])

        notBoth s1 s2 = Not (And [predicate s1,predicate s2])

        combis xs
          | xs == []  = []
          | otherwise = [(x,y) | x <- [head xs], y <- tail xs]
                        ++ combis (tail xs)

singletons = do reserved "singletons"
                colon
                ss <- commaSep lowerSymbol
                return $ AnnotatedFormula
                         "singletons" Axiom (formula ss)
  where
    formula ss = Forall [Variable "X"] (And (map neq (combis ss)))

    neq (s1,s2) = Atom (t1 `Neq` t2)
      where
        t1 = Function s1 []
        t2 = Function s2 []

    combis xs
      | xs == []  = []
      | otherwise = [(x,y) | x <- [head xs], y <- tail xs]
                    ++ combis (tail xs)


annotatedFormula :: Parser Statement
annotatedFormula = axiom <|> conjecture
  where
    axiom      = parser Axiom "axiom"
    conjecture = parser Conjecture "conjecture"

    parser r s = do reserved s
                    colon
                    f <- formula
                    return $ AnnotatedFormula
                             "unnamed" r (bindFreeVars f)

-- Formulae
formula :: Parser Formula
formula = try binary <|> unitary

binary :: Parser Formula
binary = try nonassociative <|> associative
  where
    nonassociative = try equivalence <|> implication
      where
```

```
        equivalence = parser Iff "<=>"
        implication = parser Implies "=>"

        parser :: (Formula -> Formula -> Formula) -> String
               -> Parser Formula
        parser c s = do f1 <- unitary
                        symbol s
                        f2 <- unitary
                        return $ c f1 f2

    associative = unitary `chainr1` connective
      where
        connective
          = do { symbol "&"; return (\ f1 f2 -> And [f1,f2]) }
            <|> do { symbol "|"; return (\ f1 f2 -> Or [f1,f2]) }

unitary :: Parser Formula
unitary = parensFormula <|> negation <|> quantifiedFormula <|> atom
  where
    parensFormula     = do symbol "("
                           f <- formula
                           symbol ")"
                           return f

    negation          = do reservedOp "-"
                           f <- unitary
                           return $ Not f

    quantifiedFormula = do q <- oneOf "!?"
                           symbol "["
                           vs <- commaSep variable
                           symbol "]"
                           colon
                           f <- unitary
                           return $ case q of
                                      '!' -> Forall vs f
                                      '?' -> Exists vs f

    atom              = do t <- term
                           return $ Atom t

-- Terms
term :: Parser Term
term = try equation <|> (try function <|> constant <|> variable)

equation :: Parser Term
equation = try (eq Neq "!=") <|> eq Eq "="
  where
    term = variable <|> try function <|> constant

    eq c rOp = do t1 <- term
                  op <- reservedOp rOp
                  t2 <- term
                  return $ c t1 t2

function :: Parser Term
function = do s     <- lowerSymbol
              isTC  <- option False (do { char '+'; return True })
              isRTC <- option False (do { char '*'; return True })
              symbol "("
              ts    <- commaSep term
              symbol ")"
              return $ if isTC
                       then TC (Function s ts)
                       else if isRTC
                            then RTC (Function s ts)
                            else Function s ts

constant :: Parser Term
constant = do s <- lowerSymbol
              return $ Function s []

variable :: Parser Term
variable = do s <- upperSymbol
              return (Variable s)
```

B.1.2 AbstractSyntax Module

```
module ETPTP.AbstractSyntax
  ( Statement(..)
  , Name
  , Role(..)
  , module ETPTP.Formula
  )
  where

import ETPTP.Formula

-- Statement
data Statement = Include FilePath
               | AnnotatedFormula { name    :: Name
                                  , role    :: Role
                                  , formula :: Formula
                                  }
  deriving (Eq,Show)

type Name = String

data Role = Axiom | Conjecture
  deriving (Eq,Show)
```

B.1.3 Formula Module

```
module ETPTP.Formula where

import Data.List (nub)

----------------------------------------------------------------------
-- Formulae
----------------------------------------------------------------------

data Formula = Atom Term
             | Not Formula
             | And [Formula]
             | Or [Formula]
             | Formula `Implies` Formula
             | Formula `Iff` Formula
             | Forall [Term] Formula
```

```
            | Exists [Term] Formula
  deriving (Eq,Show)

freeVars :: Formula -> [Term]
freeVars (Atom t)          = vars t
freeVars (Not f)           = freeVars f
freeVars (And fs)          = nub (concatMap freeVars fs)
freeVars (Or fs)           = nub (concatMap freeVars fs)
freeVars (f1 `Implies` f2) = nub (concatMap freeVars [f1,f2])
freeVars (f1 `Iff` f2)     = nub (concatMap freeVars [f1,f2])
freeVars (Forall ts f)     = filter (\x -> x `notElem` ts) (freeVars f)
freeVars (Exists ts f)     = filter (\x -> x `notElem` ts) (freeVars f)

bindFreeVars :: Formula -> Formula
bindFreeVars f
  | length (freeVars f) > 0 = Forall (freeVars f) f
  | otherwise               = f

-----------------------------------------------------------------------
-- Terms
-----------------------------------------------------------------------

data Term = Variable Symbol
          | Function Symbol [Term]
          | TC Term
          | RTC Term
          | Term `Eq` Term
          | Term `Neq` Term
  deriving (Show)

instance Eq Term where
  Variable s1 == Variable s2          = s1 == s2
  Function s1 ts1 == Function s2 ts2 = s1 == s2 && ts1 == ts2
  TC t1 == TC t2                      = t1 == t2
  RTC t1 == RTC t2                    = t1 == t2
  t1 `Eq` t2 == t3 `Eq` t4            = t1 == t3 && t2 == t4
                                        || t1 == t4 && t2 == t3
  t1 `Neq` t2 == t3 `Neq` t4          = t1 == t3 && t2 == t4
                                        || t1 == t4 && t2 == t3
  t1 == t2                            = False

type Symbol = String

vars :: Term -> [Term]
vars (Variable s)    = [Variable s]
vars (Function s ts) = nub (concatMap vars ts)
vars (TC t)          = vars t
vars (RTC t)         = vars t
vars (t1 `Eq` t2)    = nub (vars t1 ++ vars t2)
vars (t1 `Neq` t2)   = nub (vars t1 ++ vars t2)
```

B.1.4 BackEnd Module

```
module ETPTP.BackEnd
  ( Statement(..)
  , toTPTP
  )
  where
```

```
import Data.List (nub,nubBy)
import ETPTP.AbstractSyntax

type Arity = Int

toTPTP :: [Statement] -> [Statement]
toTPTP ss = nubBy
            (\s1 s2 -> (s1 == s2) && (not (s1 ‘elem‘ ss)))
            (concatMap convertStatement ss)
  where
    convertStatement :: Statement -> [Statement]
    convertStatement (AnnotatedFormula n r f)
      = (AnnotatedFormula n r (replaceRTCs f)) : extraStatements n f
    convertStatement s = [s]

    replaceRTCs :: Formula -> Formula
    replaceRTCs (Atom t)        = Atom (convertTerm t)
    replaceRTCs (Not f)         = Not (replaceRTCs f)
    replaceRTCs (And fs)        = And (map replaceRTCs fs)
    replaceRTCs (Or fs)         = Or (map replaceRTCs fs)
    replaceRTCs (Implies f1 f2) = Implies (replaceRTCs f1)
                                          (replaceRTCs f2)
    replaceRTCs (Iff f1 f2)     = Iff (replaceRTCs f1) (replaceRTCs f2)
    replaceRTCs (Forall vs f)   = Forall vs (replaceRTCs f)
    replaceRTCs (Exists vs f)   = Exists vs (replaceRTCs f)

    convertTerm :: Term -> Term
    convertTerm (RTC (Function s ts)) = Function (s ++ "_RTC") ts
    convertTerm (TC (Function s ts))  = Function (s ++ "_TC") ts
    convertTerm t                     = t

    extraStatements :: Name -> Formula -> [Statement]
    extraStatements n f = concatMap (axioms n) (extractDataRTCs f)

    extractDataRTCs :: Formula -> [(Symbol,Arity)]
    extractDataRTCs f
      = nub (map (\t -> (symbol t, arity t)) (extract f))
      where
        symbol :: Term -> Symbol
        symbol (Function s ts) = s

        arity :: Term -> Arity
        arity (Function s ts) = length ts

        extract :: Formula -> [Term]
        extract (Atom (TC t))   = [t]
        extract (Atom (RTC t))  = [t]
        extract (Atom _)        = []
        extract (Not f)         = extract f
        extract (And fs)        = concatMap extract fs
        extract (Or fs)         = concatMap extract fs
        extract (Implies f1 f2) = (extract f1)
                                  ++ (extract f2)
        extract (Iff f1 f2)     = (extract f1)
                                  ++ (extract f2)
        extract (Forall _ f)    = extract f
        extract (Exists _ f)    = extract f

    axioms :: Name -> (Symbol,Arity) -> [Statement]
    axioms n (s,a)
      = [tc,i1,i2,i3,e1,e2,e3,c1,c2]
      where
```

```
ws = map Variable (map (\i -> ('W':show i)) [1..a-2])
x  = Variable "X"
y  = Variable "Y"
z  = Variable "Z"
v  = Variable "V"
ne = Atom (x `Neq` y)

sTC  = s ++ "_TC"
sRTC = s ++ "_RTC"
sS   = s ++ "_S"
sC   = s ++ "_C"

function s ts = Atom (Function s ts)

tc = let vs  = ws ++ [x,y]
         vs1 = ws ++ [x,z]
         vs2 = ws ++ [z,y]

         f = Forall vs (g `Iff` h)
         g = function sTC vs
         h = Exists [z] (And [function s vs1
                             ,function sRTC vs2])
     in AnnotatedFormula
        sTC Axiom f

i1 = let vs  = ws ++ [x]
         vs' = vs ++ [x]
         f   = Forall vs (function sRTC vs')
     in AnnotatedFormula
        (s ++ "_I1") Axiom f

i2 = let vs = ws ++ [x,y]
         f  = Forall vs ((function s vs)
                   `Implies` (function sRTC vs))
     in AnnotatedFormula
        (s ++ "_I2") Axiom f

i3 = let vs  = ws ++ [x,y,z]
         vs1 = ws ++ [x,y]
         vs2 = ws ++ [y,z]
         vs3 = ws ++ [x,z]
         f   = Forall vs ((And [function sRTC vs1
                               ,function sRTC vs2])
                    `Implies` (function sRTC vs3))
     in AnnotatedFormula
        (s ++ "_I3") Axiom f

e1 = let vs = ws ++ [x,y]
         ts = ws ++ [x,Function sS vs]
         f  = Forall vs ((And [function sRTC vs, ne])
                   `Implies` (function s ts))
     in AnnotatedFormula
        (s ++ "_E1") Axiom f

e2 = let vs = ws ++ [x,y]
         ts = ws ++ [Function sS vs,y]
         f  = Forall vs ((And [function sRTC vs, ne])
                   `Implies` (function sRTC ts))
     in AnnotatedFormula
          (s ++ "_E2") Axiom f

e3 = let vs = ws ++ [x,y]
```

```
                 ts = ws ++ [x,Function sS vs,y]
                 f  = Forall vs ((And [function sRTC vs, ne])
                                  `Implies` (function sC ts))
             in AnnotatedFormula
                (s ++ "_E3") Axiom f

        c1 = let vs  = ws ++ [x,v]
                 vs' = ws ++ [x,x,v]
                 f   = Forall vs (Not (function sC vs'))
             in AnnotatedFormula
                (s ++ "_C1") Axiom f

        c2 = let vs  = ws ++ [x,y,z,v]
                 vs1 = ws ++ [x,y,v]
                 vs2 = ws ++ [y,z,v]
                 vs3 = ws ++ [x,z,v]
                 f = Forall vs ((And [function sC vs1,function sC vs2])
                                `Implies` (function sC vs3))
             in AnnotatedFormula
                (s ++ "_C2") Axiom f
```

B.1.5 PrettyPrinter Module

```
module ETPTP.PrettyPrinter (printer) where

import ETPTP.AbstractSyntax
import Text.PrettyPrint.HughesPJ

printer :: [Statement] -> String
printer ss
  = renderStyle style (vcat (map statementToDoc ss))
  where
    style = Style PageMode 64 0.5

-----------------------------------------------------------------------
-- Statements
-----------------------------------------------------------------------

statementToDoc :: Statement -> Doc
statementToDoc (Include f) = hcat (map text ["include('", f, "')."])
statementToDoc (AnnotatedFormula n r f)
  = sep [ hcat[text "fof(", space, text n, comma, space, role r, comma]
        , nest 2 (formulaToDoc f <+> text ").")
        ]
    where
      role Axiom      = text "axiom"
      role Conjecture = text "conjecture"

-----------------------------------------------------------------------
-- Formulae
-----------------------------------------------------------------------

formulaToDoc :: Formula -> Doc
formulaToDoc (Atom t)      = termToDoc t
formulaToDoc (Not f)       = (char '~') <> formulaToDoc f
formulaToDoc (Forall vs f) = quantifiedFormulaToDoc '!' vs f
formulaToDoc (Exists vs f) = quantifiedFormulaToDoc '?' vs f
formulaToDoc (And fs)
  = parens (sep (intersperse '&' (map formulaToDoc fs)))
```

```
formulaToDoc (Or fs)
  = parens (sep (intersperse '|' (map formulaToDoc fs)))
formulaToDoc (Implies f1 f2)
  = parens (sep [formulaToDoc f1, hsep [text "=>", formulaToDoc f2]])
formulaToDoc (Iff f1 f2)
  = parens (sep [formulaToDoc f1, hsep [text "<=>", formulaToDoc f2]])

intersperse :: Char -> [Doc] -> [Doc]
intersperse c ds = (head ds) : map (\d -> hsep [char c, d]) (tail ds)

quantifiedFormulaToDoc :: Char -> [Term] -> Formula -> Doc
quantifiedFormulaToDoc c vs f
  = sep [bindings, hsep [char ':', formulaToDoc f]]
    where
      bindings = char c
              <> char '['
              <> (cat (punctuate comma (map termToDoc vs)))
              <> char ']'

-----------------------------------------------------------------------
-- Terms
-----------------------------------------------------------------------

termToDoc :: Term -> Doc
termToDoc (Variable s)           = text s
termToDoc (TC (Function s ts))  = termToDoc (Function (s++['+']) ts)
termToDoc (RTC (Function s ts)) = termToDoc (Function (s++['*']) ts)
termToDoc (Eq t1 t2)      = sep [termToDoc t1, char '=', termToDoc t2]
termToDoc (Neq t1 t2)     = sep [termToDoc t1, text "!=", termToDoc t2]
termToDoc (Function s []) = (text s)
termToDoc (Function s ts)
  = (text s) <> (parens (cat (punctuate comma (map termToDoc ts))))
```

B.2 Visualisation Tool

```
module Model2Dot where

import Control.Monad.State
import Data.List (nub)
import System.IO
import Text.ParserCombinators.Parsec hiding (State,Line)
import Text.Regex

main = do input <- hGetContents stdin
          hPutStrLn stdout (toDot (lines input))

-----------------------------------------------------------------------
-- Printing and Conversion
-----------------------------------------------------------------------

toDot :: [String] -> String
toDot ls = g
  where
    ls'= (map parseLine (filter (\x -> x /= "") (map preprocess ls)))

    g = "digraph g {\n\tgraph [rankdir=LR];\n"
        ++ concatMap node (nodes (subRhs ls'))
        ++ concatMap edge (edges (subRhs ls'))
        ++ "}\n"


    node :: Node -> String
    node (o,ss)
      = "\tnode" ++ o ++ " [label=\"" ++ o
        ++ "|{" ++ label ++ "}\",shape=record];\n"
      where
        label
          | length ss == 1 = head ss
          | otherwise      = foldl1 (\l1 l2 -> l1 ++ "|" ++ l2) ss

    edge :: NumEdge -> String
    edge ((os,s),n)
      = "\t" ++ edge ++ " [label=\"" ++ s ++ label ++"\"];\n"
      where
        edge = foldl1 (\n1 n2 -> n1 ++ " -> " ++ n2)
                      (map (\n -> "node"++ n) os)

        label
          | length os > 2 = " (#" ++ (show n) ++ ")"
          | otherwise     = ""

type Node    = (Object,[String])
type Edge    = ([Object],String)
type NumEdge = (Edge,Int)

nodes :: [Eqv] -> [Node]
nodes eqvs = map ss (nub (map fst ns))
  where
    ns = map node (filter test eqvs)
    test (Eqv (_,os) ("$true",_)) = length os == 1
    test _                        = False
    node (Eqv (s,[o]) _) = (o,[s])
    ss o = (o, concatMap snd (filter (\x -> o == fst x) ns))

edges :: [Eqv] -> [NumEdge]
edges eqvs = numEdges es
  where
    es = map edge (filter test eqvs)
    test (Eqv (_,os) ("$true",_)) = length os > 1
    test _                        = False
    edge (Eqv (s,os) _) = (os,s)

    numEdges :: [Edge] -> [NumEdge]
    numEdges es = evalState (numEdges es) []
      where
        numEdges :: [Edge] -> State [String] [NumEdge]
        numEdges [] = return []
        numEdges (e:es) = do ne  <- numEdge e
                             nes <- numEdges es
                             return (ne:nes)

        numEdge e@(os,s) = do state <- get
                              newState <- return (s:state)
                              put newState
                              return $ (e,count s state)

        count x ys = length (filter (\y -> y == x) ys)


------------------------------------------------------------------------
```

```
-- Right-Hand Side Substitution
----------------------------------------------------------------------

subRhs :: [Eqv] -> [Eqv]
subRhs eqvs = evalState (subEqvs eqvs) []
  where
    subEqvs :: [Eqv] -> State [(Symbol,[Object])] [Eqv]
    subEqvs [] = return []
    subEqvs (eqv:eqvs) = do e  <- subEqv eqv
                            es <- subEqvs eqvs
                            return (e:es)
      where
        subEqv eqv = do state <- get
                        (newState,newEqv) <- return (sEqv eqv state)
                        put newState
                        return newEqv

        sEqv eqv@(Eqv l r) state
          | r == ("$true",[]) = (l:state,eqv)
          | r `elem` state    = (state,Eqv l ("$true",[]))
          | otherwise         = (state,Eqv l ("$false",[]))


----------------------------------------------------------------------
-- Parsing
----------------------------------------------------------------------

parseLine :: String -> Eqv
parseLine s
  = case (parse line "" s) of
      Left  e -> error (show e)
      Right l -> l

data Eqv = Eqv (Symbol,[Object]) (Symbol,[Object])
  deriving (Eq,Show)

type Symbol = String
type Object = String

line :: Parser Eqv
line = try equation <|> equivalence
  where
    equation = do s <- symbol
                  ns <- objects
                  char '='
                  n <- symbol
                  return $ Eqv (s,ns++[n]) ("$true",[])

    equivalence = do s1  <- symbol
                     ns1 <- objects
                     string "<=>"
                     s2  <- symbol
                     ns2 <- objects
                     return $ Eqv (s1,ns1) (s2,ns2)


    symbol  = many1 (alphaNum <|> char '_' <|> char '$')
    objects = do many (char '(')
                 ns <- sepBy symbol (char ',')
                 many (char ')')
                 return ns
```

```
--------------------------------------------------------------------------
-- Preprocessing
--------------------------------------------------------------------------

preprocess :: String -> String
preprocess l = l'
  where
    re = mkRegex "(.*)(<=>|=!)(.*)"
    l' = case matchRegex re (foldl (++) "" (words l)) of
           Nothing -> ""
           Just ss -> filter (\x -> x /= '!') (foldl1 (++) ss)
```