An Object-Oriented Implementation of a KE Tableau Prover

Adolfo Gustavo Serra Seca Neto

December 11, 2003

Abstract

The implementation of Automated Theorem Provers (ATP's) has been a research theme in Computer Science for a long time. ATP's have many applications and some important results were obtained with them. But the implementation of tableau based theorem provers has to be better developed, because tableau provers are usually considered to be less efficient than Resolution-based provers. The KE system [16] is an improvement, in the computational sense, over traditional Smullyan's Analytic Tableaux [75], that is more 'natural' and more efficient than Smullyan's tableaux. Tableau-based theorem provers can be very useful for studying properties of logical deduction, specially if we can give a good modularization of proof strategies. Object-orientation, aspect-orientation and other related techniques can help us better modularize proof strategies. That is the the main objective of our research: to achieve a better modularization of strategies in tableau provers. Having this in mind, we are incrementally developing a KE Tableau Prover. We have just finished our first version, an object-oriented KE tableau prover, which we will present here.

1 Introduction

The implementation of Automated Theorem Provers (ATP's) has been a research theme in Computer Science for a long time. ATP's have many applications and some important results were obtained with them. But there is still much to be done in this area. For instance, the implementation of tableau based theorem provers has to be better developed, because tableau provers are usually considered to be less efficient than Resolution-based provers. The Tableau method is a formal proof procedure that has many variants and exists for several logics [28]. One of these variants is the KE system [16], which was presented as an improvement, in the computational sense, over traditional Smullyan's Analytic Tableaux [75]. That is, the KE System is a 'natural' proof method and it is more efficient than Smullyan's tableaux.

Tableau-based theorem provers can be very useful for studying properties of logical deduction, specially if we can give a good modularization of proof strategies. Object-orientation, aspect-orientation and other related techniques can help us better modularize proof strategies. That is the the main objective of our research: to achieve a better modularization of strategies in tableau provers. Having this in mind, we are incrementally developing a KE Tableau Prover. We have just finished our first version, an object-oriented KE tableau prover, which we will present here.

1.1 Outline

In section 2 we will present automated theorem provers (ATP's) and automated reasoning (AR). ATP's applications, results and some other issues related to ATP's and AR will be briefly discussed. Then, in section 3 we will describe our project: an investigation into the construction of Tableau based Automated Theorem Provers using object-orientation, design patterns, aspect-orientation, and some extreme programming techniques. After that, in section 4 we will describe with some diagrams and texts our current implementation.

Test results comparing the performance of our system with an existing system are presented in section 5. Then some future works will be discussed in section 6: refactoring (both object-oriented and aspect-oriented), the writing of analysis patterns for ATP's, of a component-based architecture for Tableau Provers and the development of a framework. In section 7 we shall present our concluding remarks.

2 Automated Theorem Provers and Automated Reasoning

The implementation of Automated Theorem Provers (ATP's) has been a research theme in Computer Science for a long time. The history of this field dates back to the 1950´s. Read below an excerpt of [14]:

> *Automatic deduction,* or mechanical-theorem proving, has been a major concern of AI since its earliest days. At the first formal conference on AI, held at Dartmouth College in the summer of 1956, Newell and Simon (1956) discussed the Logic Theorist, a deduction system for propositional logic. Misnky was concurrently developing the ideas that were later embodied in Gelernter's theorem prover for elementary geometry (see McCorduck, 1979, p. 106; Gelernter, 1963). Shortly after this, Wang (1960) produced the first implementation of a reasonably efficient, complete algorithm for proving theorems in propositional logic.

ATP's have many applications, among which we highlight the following [15]:

- proof of mathematical theorems
  - proof verification: to detect and to correct errors in proofs
  - theorem proving: to find new proofs automatically
  - helping the teaching of mathematics
- suuport for designing reliable software
  - verification: to find errors or prove correction
  - optimization: to increase performance
  - synthesis: to generate code from specifications
- inference engine for Artificial Intelligent systems
  - general problem solvers
  - robot planning
  - intelligent agents

In [69], one can find that "the first published results from research on automated deduction using electronic computers were those of Newell, Shaw, and Simon (1957) on the Logic Theorist. This program was based on an attempt to model human thought processes. Martin Davis (1957) had actually designed a program that came up with a proof in 1954, but the Logic Theorist's results were published slighty earlier. Both Davis's 1954 program and the Logic Theorist were based on somewhat ad hoc methods that did not strongly influence later automated deduction.

(...)

Another use of theorem provers is as an assistant, providing advice to, say, a mathematician. In this mode the mathematician acts as a supervisor, mapping out the strategy for determining what to do next and asking the theorem prover to fill in the details. This alleviates the problem of semi-decidability to some extent, bacause the supervisor can cancel a query and try another approach if the query is taking too much time. A theorem prover can also act as a **proof-checker**, where the proof is given by a human as a series of fairly large steps; the individual inferences required to show that each step is sound are filled in by the system."

Some theorem provers have even come up with new mathematical results, such as the settling of the Robbins problem by the ATP System EQP in 1996. The Robbins problem remained open for 63 years.

The field of Automated theorem-proving (ATP) is also known as:

- Automated deduction [69]
- Automatic deduction [14]
- Mechanical theorem-proving [13, 14]
- Automated reasoning [82].

And theorem provers [69] are also known as

- Automated reasoners [69]
- Theorem-proving systems [61].

However, in my opinion the use of the word reasoning is delicate. Reasoning is an activity that can only be performed by human beings. In its beginnings, the field of AI had too much expectations. For instance, according to [69] (page 17), in a workshop at Dartmouth in 1956, Allen Newell and Herbert Simon, two researchers from Carnegie Tech, presented a reasoning program, the Logic Theorist (LT), about which Simon claimed, "We have invented a computer program capable of thinking non-numerically, and thereby solved the venerable mind-body problem." This was clearly an exaggeration, although this program later was able to prove most of the theorems in Chapter 2 of Russell and Whitehead's *Principia Mathematica.*

But the definition of reasoning in the area of automated reasoning is much more modest. For example, [82] writes the following:

> *Reasoning* is the process of drawing conclusions from facts. For the reasoning to be sound, these conclusions must follow inevitably from the facts from which they are drawn. (...)
>
> The object of *automated reasoning* is to write computer programs that assist in solving problems and in answering questions requiring reasoning. The assistance provided by an automated reasoning program is available in two different modes. You can use such a program in an interactive fashion, that is, you can instruct it to draw some conclusions and present them to you, and then it will ask you for further instructions. Or you can use such a program in a batch mode, that is, you can assign it an entire reasoning task and await the final result.

The implementation of ATP's has been discussed in several books such as [82] and [27]. Nowadays there are several ATP's based on different proof techniques (Resolution, Natural Deduction, Tableaux) and implemented in distinct languages (C, Prolog, Java, among others). A broad list of known systems can be found in [49]. Competitions are held amongst these systems [77] that use databases of problems such as [77, 71].

The *Journal of Automated Reasoning* is the principal journal for theorem proving. And the major conference in this field is the annual International Conference on Automated Deduction (CADE), as informed by [69].

3 Project

In our work we are investigating the construction of Tableaux-based Automated Theorem Provers. We are going to use object-orientation, design patterns, aspect-orientation, and some extreme programming techniques. Below we shall explain each of these items. The result will be a tableau prover for classical propositional logic. It will be implemented in Java and AspectJ and during its development we intend to use, whenever we feel it is adequate, design patterns and some extreme programming techniques.

Two tableaux-based automated theorem provers will serve as a base for our studies: the system presented in [20] and jTAP [6]. Besides being tableau-based, both were implemented in object-oriented programming languages.

3.1 Tableaux

The Tableaux method is a formal proof procedure that has many variants and exists for several logics [28]. Tableau methods have been found to be a convenient formalism for automating deduction in various non-standard logics as well as in classical logic [79]. It was with Smullyan [75] that tableaux became well-known in the logic community. In his book he presented a system called "Analytic Tableaux". The tableaux method is a refutational procedure. That is, in order to prove that a formula X is valid we try to show that it is not valid. Having this in mind, we apply a procedure for inferring the logical consequences of the formulas present in the tableaux. If any branch of the generated proof tree remains open, then we have a refutation of X.

3.1.1 The KE System

Marco Mondadori and Marcello D'Agostino presented the KE System [16], a refutation system that, they argued, though "being close to the tableau method, is not affected by the anomalies of cut-free systems." KE Tableaux are very similar to Analytic Tableaux because they also use signed formulas, have expansion rules with premises and conclusions and have the same rules for the construction of the initial tableau and the closing of the tableau.

D'Agostino proposed KE Tableaux as an improvement, in the computational sense, over Smullyan's Analytic Tableaux. He proved that a canonical restriction of the KE System can polynomially simulate the tableau method but the tableau method cannot polinomially simulate the canonical restriction of KE[1]. That is, the KE System is a 'natural' proof method and it is more efficient than Smullyan's tableaux.

The set of rules for the KE System (see Figure 1) has only one rule of the branching type, the PB rule, corresponding to the principle of bivalence. With this rule, in conjunction with the usual tableau rules,

[1]In other words, for each analytic tableau proof of size n, there is a KE-Tableau proof with size polynomial in n. But there is at least one proof in KE Tableau of size n whose corresponding proof in Analytic Tableaux has size superpolynomial in n.

it is not difficult to construct short refutations of the 'hard examples' showed for Smullyan's tableaux in [18].

Disjunction Rules

$$\frac{\begin{array}{l}T\,A\vee B\\F\,A\end{array}}{T\,B}\ (\mathrm{E}T\vee 1)\qquad \frac{\begin{array}{l}T\,A\vee B\\F\,B\end{array}}{T\,A}\ (\mathrm{E}T\vee 2)\qquad \frac{F\,A\vee B}{\begin{array}{l}F\,A\\F\,B\end{array}}\ (\mathrm{E}F\vee)$$

Conjunction Rules

$$\frac{\begin{array}{l}F\,A\wedge B\\T\,A\end{array}}{F\,B}\ (\mathrm{E}F\wedge 1)\qquad \frac{\begin{array}{l}F\,A\wedge B\\T\,B\end{array}}{F\,A}\ (\mathrm{E}F\wedge 2)\qquad \frac{T\,A\wedge B}{\begin{array}{l}T\,A\\T\,B\end{array}}\ (\mathrm{E}T\wedge)$$

Implication Rules

$$\frac{\begin{array}{l}T\,A\rightarrow B\\T\,A\end{array}}{T\,B}\ (\mathrm{E}T\rightarrow 1)\qquad \frac{\begin{array}{l}T\,A\rightarrow B\\F\,B\end{array}}{F\,A}\ (\mathrm{E}F\rightarrow 2)\qquad \frac{F\,A\rightarrow B}{\begin{array}{l}T\,A\\F\,B\end{array}}\ (\mathrm{E}F\rightarrow)$$

Negation Rules

$$\frac{T\,\neg A}{F\,A}\ (\mathrm{E}T\neg)\qquad \frac{F\,\neg A}{T\,A}\ (\mathrm{E}F\neg)$$

Principle of Bivalence

$$\frac{}{T\,A\mid F\,A}\ (\mathrm{PB})$$

Figure 1: KE tableau expansion rules

Besides that, the KE System introduces simple elimination rules of linear type which combined with PB yield a refutation system for classical logic that is complete and sound. And the system can be restricted to analytic applications of PB, i.e. applications which do not violate the subformula property, and remain complete and sound.

3.1.2 Nondeterminism in the KE system

In [18], a simple but complete refutation procedure for the KE system, the so-called *canonical procedure* is presented. But this procedure "is not, strictly speaking, a completely deterministic algorithm. Some steps involve a choice and different strategies for making these choices will lead to different algorithms." For instance, for some of the problems used in the tests of section 5, this choice can lead to linear or exponential proofs of the same set of formulas. Therefore, strategies can be more important than data structures and programming languages when one is implementing a tableau prover.

3.2 Object Orientation

The object-oriented (OO) development paradigm is regarded nowadays as the dominant software development paradigm. An important objective of OO is reuse. Reuse of code as well as resuse of analysis and design artifacts. In order to achieve this, several basic concepts are present in OO modeling and programming languages, such as inheritance, encapsulation, abstraction and composition. The most widely used modeling language, UML (*Unified Modeling Language*) [19], as well as the more popular programming languages (such as C++ and Java) are based on this paradigm.

3.2.1 Design Patterns

Since their appearance in [33], design patterns have become one of the most important tools for OO systems designers. Their development was motivated by the following observation: "designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder" [33]. Design patterns are abstract descriptions of design solutions that appear repeatedly in well-designed systems. They make it easier to reuse successful architecture and design artifacts. The [33] book has presented 23 design patterns and in the literature nowadays one can find many more design patterns for different kinds of application, as well as other kinds of patterns, such as architectural patterns [12, 72] and analysis patterns [29].

3.3 Aspect Orientation

The object oriented paradigm (OOP) has some limitations such as the scattering and tangling of code with different purposes. Part of these limitations can be overcome with the use of techniques of design patterns. Aspect oriented programming (AOP) is an attempt to solve these and other problems identified in the object oriented paradigm. It is a technique that intends to achieve more modularity in situations where object orientation and the use of design patterns is not enough.

The main motivation for the development of AOP was the alleged inability of OOP and other current programming paradigms to fully support the *separation of concerns* principle [46]. According to AOP proponents, AOP solves some problems of OOP by allowing an adequate representation of the so-called *crosscutting concerns* [76]. With AOP, code that implements crosscutting concerns, i.e. that implements specific functions that affect different parts of a system, and would be scattered and tangled in an OOP implementation, can be localized, increasing modularity. With this increase in modularity, one can achieve software that is more adaptable, maintainable and evolvable in the face of changing requirements.

There are several expected benefits of using AOP [65]:

- more readable and reusable code;

- a more natural mapping of system requirements to programming constructs;
- software that is more adaptable, maintainable and evolvable in the face of changing requirements.

It is also assumed that there will be an improvement in the comprehension and maintanability of complex programs by localizing behaviors (*concerns*) that would otherwise be scattered and tangled [56].

AspectJ seems to be the most promising approach to adding support for aspects to Java. At least, it is regarded as the most mature approach at the time of this writing. AspectJ is a general purpose, seamless aspect-oriented extension to the Java language that enables the modular implementation of a wide range of crosscutting concerns [3]. Therefore, an AspectJ program is a Java program with some extra constructs. These language constructs included allow the implementation of AOP features such as pointcuts, advice, inter-type declarations and aspects. Pointcuts and advice dynamically affect program flow, while inter-type declarations statically affect the class hierarchy of a program.

The modularization unit in AspectJ is called an *aspect*, just as a common concern's implementation in OOP is called a *class*. In AspectJ, aspects are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations [46]. Aspects are defined by aspect declarations, which have a form similar to that of class declarations. Aspects can implement interfaces and extend other aspects. However, aspects have no constructors and cannot be instantiated. Aspect declarations may include pointcut declarations, advice declarations, inter-type declarations as well as other kinds of declarations permitted in class declarations.

3.4 Extreme Programming

Extreme Programming (XP) is a lightweight (or agile) methodology that focuses on coding as the main task in software development [38]. Amongst the so-called "twelve practices of XP" presented in [38], we are interested in the following:

1. "Simple design": The idea behind "simple design" is to keep the code simple. The simplest design possible does not try to solve future problems by guessing future needs.
2. "Testing": The practice of testing is key to XP. And the tests should be automated so that the code can be modified and refactored safely, without the inclusion of errors in the system.
3. "Refactoring": The act of refactoring enables developers to add features while keeping the code simple. The idea is to not duplicate code nor write ugly, smelly code. The act of refactoring centers on testing to validate that the code still functions. Refactoring is not limited to when you need to the addition of features—refactoring can be done in order to make the code easier to understand.

3.5 Objective

The main objective of our research is to investigate if proof strategies for tableau provers can be well modularized by using object-oriented and aspect-oriented software development. Having this in mind, we are incrementally developing a KE Tableau Prover. Right now we have just finished our first version: a very simply designed KE Tableau prover for classical propositional logic.

For our development, we are using a number of tools. Below are the ones that we regard as most improtant:

Eclipse is an integrated development environment (IDE) that has support for Java, AspectJ and JUnit. It also enables some automation of refactorings

JDT - Java Development Tools is a set of tools for working with Java, an multiplatform, object-oriented programming language, in Eclipse;

AJDT - AspectJ Development Tools is a set of tools for programming in AspectJ (an aspect-oriented extension to Java) in Eclipse;

JUnit is framework for automating tests for Java programs;

JFlex is a lexical analyzer generator for Java written in Java. It is based on the well-known **lex** lexical analyzer;

CUP (Java(tm) Based Constructor of Useful Parsers) is a system for generating LALR parsers from simple specifications. It serves the same role as the widely used program YACC and in fact offers most of the features of YACC. However, CUP is written in Java, uses specifications including embedded Java code, and produces parsers which are implemented in Java [39].

4 Implementation details

Our implementation efforts are still in the beginning. There is much more to be done. We are trying to keep the design of the classes as simple as possible; we are following some guidelines for designing flexible object-oriented systems [66]. For instance, we are avoiding to add unnecessary flexibility too early. This is also advocated by XP, as well as incremental development, test-driven programming (writing the tests before writing a class) and refactoring, which we are also doing whenever possible. Finally, we have tried to apply design patterns to solve some problems that have appeared in the beginning of the development.

In Figure 6 we depict only what is essential for our design; more information can be found in the source code. A KETableau object can be instantiated using one of two constructors:

- the first one for an unsigned formula:
  `KETableau(Formula f, FormulaFactory ff);`
- the second for a set of signed formulas:
  `public KETableau(SignedFormulaList sfl, SignedFormulaFactory sff, FormulaFactory ff).`

A problem can consist of one unsigned formula or a set of signed formulas. If it is an unsigned formula X, then the signed formula FX is added to the `KETableau`. In our test cases we can see example calls of these constructors. It is important to notice that one or two factories have to be passed to the `KETableau` object. These factories are the result of an incomplete[2] application of the Flyweight design pattern [33] to the creation of formulas and signed formulas. With the use of this pattern, the system does not create two different instances of the same formula in a tableau, even if that formula appears in different branches.

The factory objects are created and updated during the analysis (lexing and parsing) of the problem that gives origin to the (signed) formulas. This analysis was implemented by using two tools: JFlex [48] and CUP [39] (see subsection 3.5). We have written two[3] grammars: one for the SAT format in [70] and other for the format used in [20].

After creating a KETableau object, we have to call the `close()` method in order to try to arrive at a proof of the problem. If it does close the tableau, then the method `isClosed()` will return true. Otherwise, it will return false.

[2] Incomplete because the (signed) formulas do not have an extrinsic state.

[3] Actually four, two for JFlex and two for CUP.

Each KETableau has one proof tree. This proof tree object includes a list of signed formulas (the signed formula or formulas passed to the tableau constructor) and possibly pointers to its successors, but no parent. Initially, the `left`, `right` and `parent` pointers are set to `null`. The `left` and `right` will be both assigned to new proof tree objects if and when we apply a PB rule (when we `addLeft` and `addRight` are used). These new proof tree objects will receive have as `parent` the proof tree that gave origin to them.

Everytime a rule is applied, a formula is added to the list of signed formulas of the current proof tree. It is based on this that each proof tree (that is in fact a branch of the tableau) is set to closed or not.

Although nary connectives are allowed (because they are used in [20] examples), we have only binary branching rules.

A SignedFormula formula object consists of a pointer to a FormulaSign object (for classical logic there are only two signs: **T** for true and **F** for false) and a pointer to a Formula. Therefore, each Formula object can give origin to 0 to 2 Signed Formula objects (in the classical case).

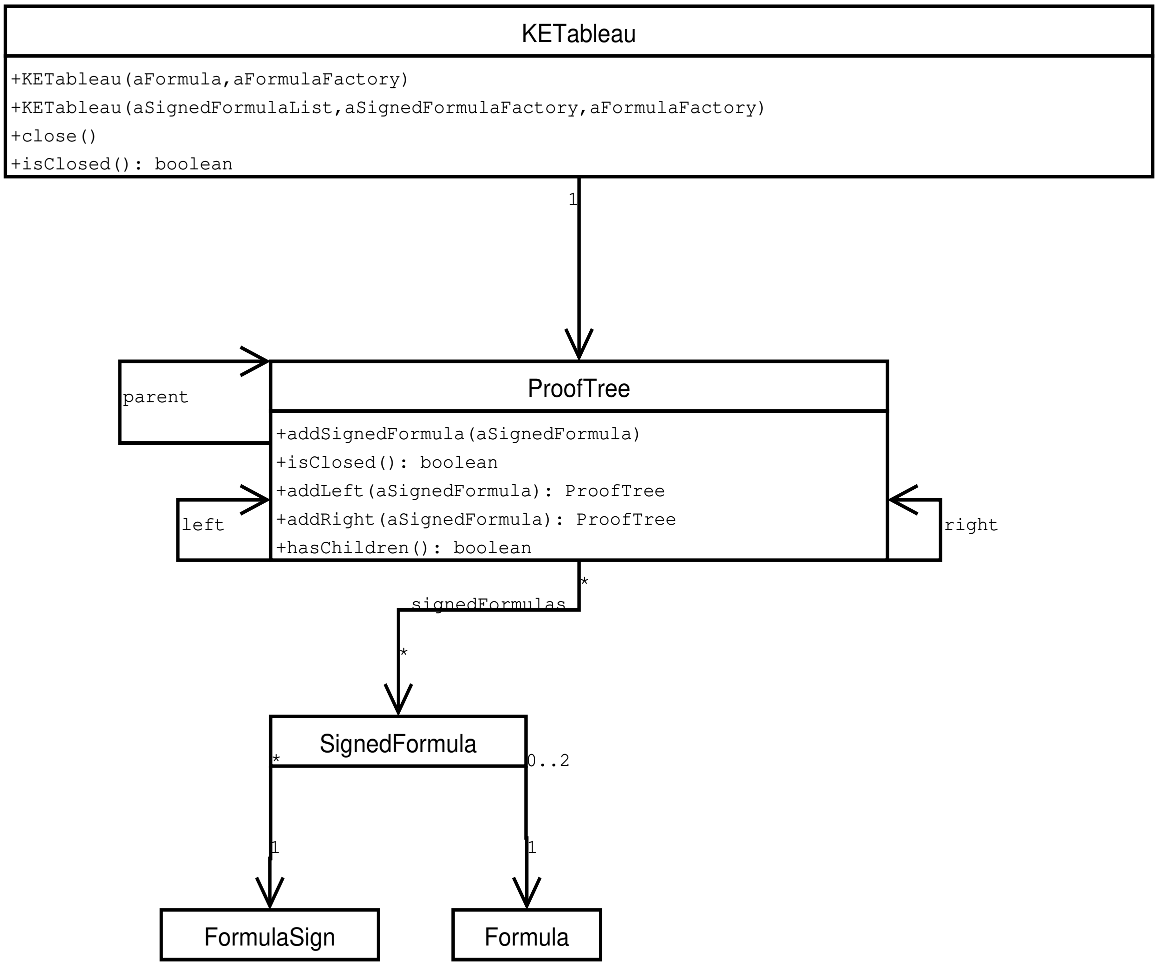

Figure 2: Simplified class diagram of the KE Tableau Prover.

We have four other diagrams that can help one to understand the source code. The classes for formulas use the Composite design pattern [33].

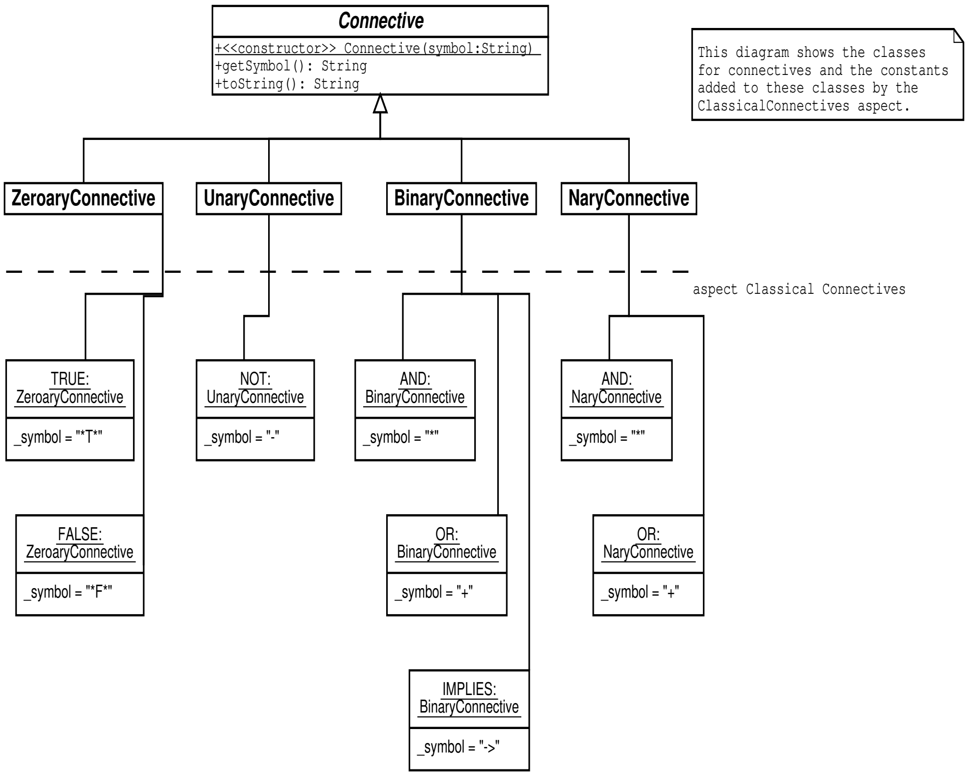

Figure 3: Class diagram for classical propositional logic connectives.

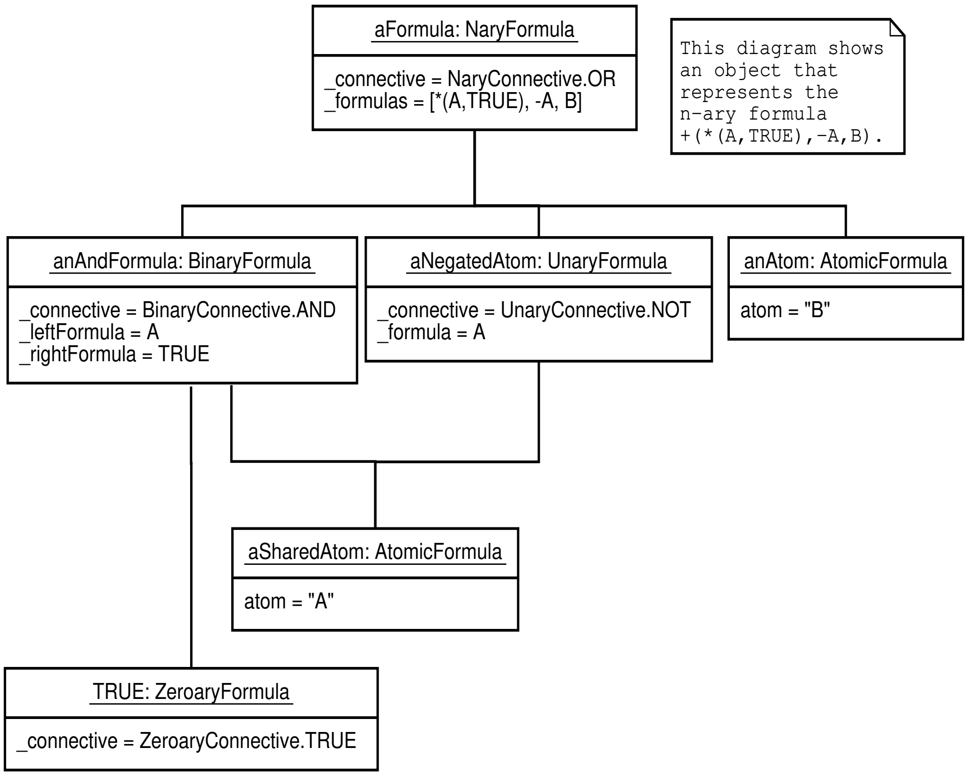

Figure 4: Object diagram for classical propositional logic formulas and connectives.

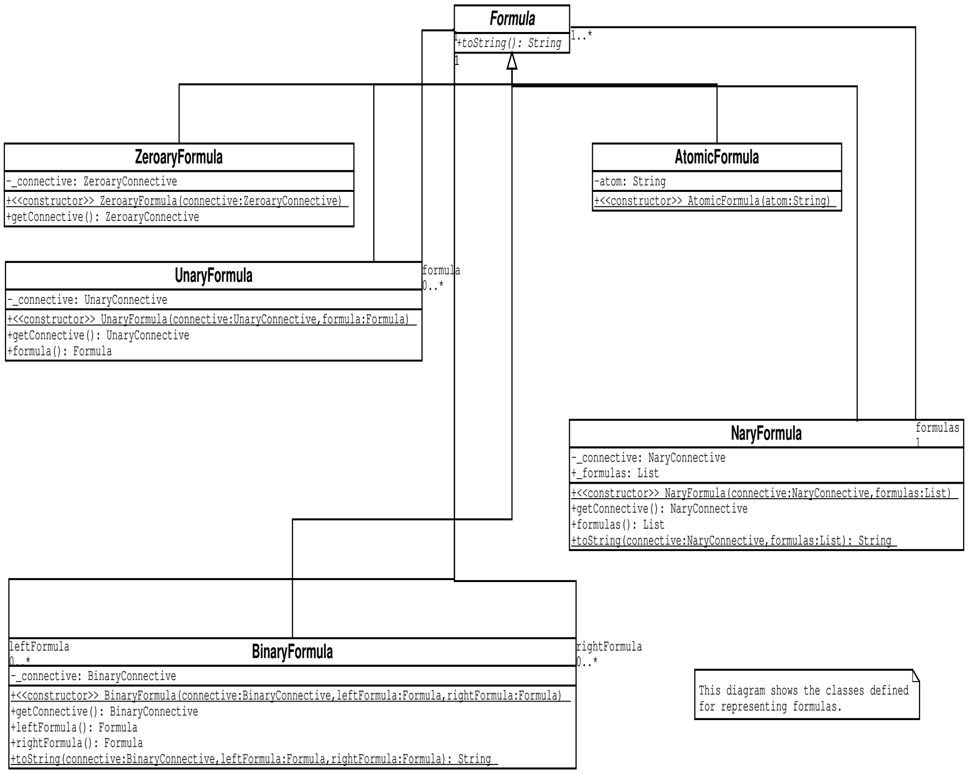

Figure 5: Class diagram for classical propositional logic formulas.

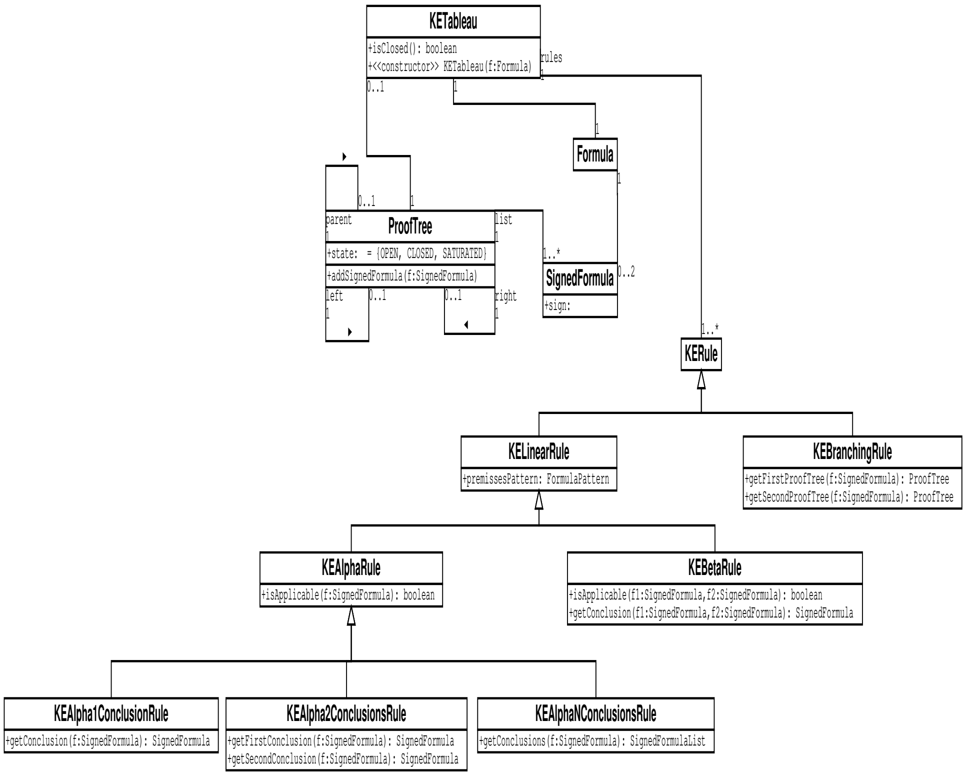

Figure 6: Class diagram for classical propositional logic rules.

5 Test results

We have tested our implementation and an implementation of the KE method [20] with some instances of the families of formulas presented in that work. In the tables below, KETP stands for our KE tableau prover, while DFTP[4] stands for the implementation of the KE method with heuristics K_i in the system described in [20].

The results were obtained in a personal computer running Linux. It is important to notice that KETP was implemented in Java and DFTP in C++. Other important aspect is that KETP currently only supports the KE method, whilst DFTP was built using a framework for many tableau methods.

For Γ, Γ_n and Statman formulas, KETP runs faster and, more important than that, requires less nodes and formulas. This is due to the fact that DFTP closes only on atomic formulas, whilst our system closes on any kind of formula. For H, H_n, PHP and Statman$_n$ fornulas, although our system required less nodes and formulas, DFTP was faster. For PHP$_n$ fornulas, DFTP was generally faster, but the analysis based on the number of nodes and formulas does not show a pattern.

5.1 Γ formulas

	KETP			DFTP		
Instance	# nodes	# formulas	Time (s)	# nodes	# formulas	Time (s)
Γ_1	1	9	0.054000	3	13	0.000479
Γ_2	3	17	0.080000	13	42	0.005284
Γ_3	7	31	0.060000	37	117	0.024991
Γ_4	15	57	0.041000	109	338	0.115548
Γ_5	31	107	0.043000	325	997	0.289457
Γ_6	63	205	0.119000	973	2970	1.053867
Γ_7	127	399	0.199000	2917	8885	4.057999
Γ_8	255	785	0.447000	8749	26626	15.474049
Γ_9	511	1555	0.909000	n/a	n/a	n/a
Γ_{10}	1023	3093	2.001000	n/a	n/a	n/a
Γ_{11}	2047	6167	4.527000	n/a	n/a	n/a

5.2 Γ_n formulas

	KETP			DFTP		
Instance	# nodes	# formulas	Time (s)	# nodes	# formulas	Time (s)
$\Gamma_n 1$	1	11	0.002000	9	26	0.001247
$\Gamma_n 2$	9	29	0.007000	23	72	0.035664
$\Gamma_n 3$	25	63	0.023000	65	207	0.058102
$\Gamma_n 4$	57	129	0.077000	191	608	0.152668
$\Gamma_n 5$	121	259	0.127000	569	1807	0.527771
$\Gamma_n 6$	249	517	0.280000	1703	5400	2.079130
$\Gamma_n 7$	505	1031	0.450000	5105	16175	7.934750

[4]Here callled Dias-Finger Tableau Prover.

5.3 H formulas

	KETP			DFTP		
Instance	# nodes	# formulas	Time (s)	# nodes	# formulas	Time (s)
H_1	1	4	0.000000	1	4	0.000068
H_2	3	15	0.002000	3	17	0.000456
H_3	7	49	0.117000	21	103	0.006212
H_4	15	197	0.601000	105	677	0.156424
H_5	31	973	10.811000	465	4777	2.895325

5.4 H_n formulas

	KETP			DFTP		
Instance	# nodes	# formulas	Time (s)	# nodes	# formulas	Time (s)
H_{n1}	1	5	0.000000	1	6	0.000094
H_{n2}	5	20	0.009000	5	21	0.000507
H_{n3}	19	81	0.139000	27	113	0.015503

5.5 PHP formulas

	KETP			DFTP		
Instance	# nodes	# formulas	Time (s)	# nodes	# formulas	Time (s)
PHP_1	1	3	0.001000	1	6	0.000116
PHP_2	3	26	0.013000	3	29	0.001226
PHP_3	11	116	0.217000	61	261	0.121501
PHP_4	63	963	3.388000	483	2219	3.685126

5.6 PHP$_n$ formulas

	KETP			DFTP		
Instance	# nodes	# formulas	Time (s)	# nodes	# formulas	Time (s)
PHP_{n1}	1	4	0.000000	3	16	0.000340
PHP_{n2}	13	65	0.109000	19	92	0.035145
PHP_{n3}	263	819	3.476000	183	812	0.284629
PHP_{n4}	2265	8846	113.016000	2147	9860	15.200194

5.7 Statman formulas

	KETP			DFTP		
Instance	# nodes	# formulas	Time (s)	# nodes	# formulas	Time (s)
$Statman_1$	1	2	0.000000	1	5	0.000092
$Statman_2$	7	18	0.001000	7	30	0.001055
$Statman_3$	15	57	0.046000	31	124	0.038595
$Statman_4$	39	162	0.088000	207	711	0.138840
$Statman_5$	87	435	0.435000	1079	3696	1.017628
$Statman_6$	183	1236	1.833000	5391	19539	8.911378

5.8 Statman$_n$ formulas

	KETP			DFTP		
Instance	# nodes	# formulas	Time (s)	# nodes	# formulas	Time (s)
$Statman_{n1}$	1	2	0.000000	1	5	0.000096
$Statman_{n2}$	7	30	0.003000	7	48	0.001506
$Statman_{n3}$	19	101	0.076000	51	276	0.045459
$Statman_{n4}$	43	266	0.259000	243	1211	0.161938

6 Future works

There is much more to do in our effort towards the implementation of tableau provers. Here we highlight some areas for future works.

6.1 Refactoring

In order to achieve better results, we need to refactor the architecture, the design and the implementation of our system. Some points for immediate attention are:

- in the source code, there are many targets for refactoring according to the catalog presented in [31];
- currently there are two versions of the system: one for Windows and other for Linux. The differences are minor: problems with the '/' and '\' signs and with the AspectJ configuration in the Linux version of Eclipse. I had to make a version for Linux without the aspect for the truth-table prover[5];
- I had problems trying to make a Java ARchive (JAR) file (a package containing several compressed .class files) containg all classes of the system because I am using computational reflection in the `parsers` package;
- the classes in the `parsers` package need to be rewritten in order to become even more general;
- the class hierarchy for KE rules is too complex. Maybe we can use interfaces instead of subclasses;
- the test cases for some classes must be completed.

After refactoring, I plan to do some additions to our system. First we will try to localize all decisions related to the proof strategy in a class or an aspect, probably by using the Strategy design pattern [33], such as [20] has done. Strategies will affect many classes in our system, therefore they are a natural candidate for aspect-orientation. For instance, a strategy might define:

- the order of application of rules or, more generally, to decide which is the next rule to be applied in a given tableau at a specific moment;
- if a branch of the tableau is going to be closed when one finds TX and FX and X is any kind of formula (this leads to shorter proofs) or only when it is an atomic formula (this simplifies the checking–it was done this way in [20]);
- when to check that a branch is closed. In the current system, it is done every time a formula is added to the tableau. But for tableaus with many formulas this can be costly. It could be done once for each three formulas added, for example, or just before the application of a PB rule;
- the data structures used for the lists of signed formulas and for the proof tree.

We are going to build both a proof checker and a proof viewer. The proof checker will enable us to check if a proof is correct more rapidly and with greater accuracy. And the proof viewer shall enable us to browse the proof and see how each formula was added to the tableau (i.e. the rule and the premiss(es) that led to that formula, if it is not a formula of the problem), among other things.

We will try to make the best possible use of OO features and design patterns but, at the same time, we are going to try to refactor for aspects [63], that is, to try to use aspect-orientation for a

[5]I have built an aspect-oriented truth-table prover in order to test the design of formulas and connectives. It uses an aspect-oriented implementation of the Interpreter pattern [33].

better modularization of strategies and the overall system. For instance, we are going to study the possibility of giving an AO implementation of the Flyweight pattern, that is, to try to remove references to FormulaFactory and SignedFormulaFactory in the code.

6.2 Analysis Patterns

According to [29], "many books on object modeling talk about analysis and design. There is little agreement on where the boundary between these two activities lies." So, what is an analysis model? In [30], he answers that analysis is about understanding how the domain works and object-oriented analysis is building a description of the domain that increases our understanding of that domain, in order to help us build an object-oriented software system. So, there is an important place in software development for analysis and analysis patterns.

Logic has been analysed for centuries. In the beginning of our development, when we were designing the classes for representing classical propositional formulas and connectives, we first thought of considering the two constants (true and false) as constant atomic formulas. But logic textbooks such as [27] classify these constants as zeroary connectives. Prof. Marcelo Finger pointed out this to me and then I changed my design. To my surprise, the design of other classes that depended upon these constants became much more clear.

We think we already have satisfactory analyzed classical propositional formulas and connectives. But we are not satisfied with the analysis for KE Tableau rules. And we have not yet begun to analyse proof strategies.

6.3 Software architecture and Component-based systems

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationship among them [4]. In other words, it is the overall system structure, describing its components and its interrelationships.

According to [78], "software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system." Modern computation systems require evolving software that is built from existing software components, developed by independent sources [9]. An initial draft of the architecture of our system is depicted in Figure 7 and following is a description of its components:

> The *Problem Analyser* component receives two inputs: *(i)* a problem description grammar (such as the ones described in TPTP [77] or in [70]) from the *Configurator* component and *(ii)* a problem description in that grammar (e.g. the ones in TPTP or SATLIB) from the *User Interface* component. Then it outputs a problem object for the *Prover* component.
>
> The *Prover* then tries to prove that problem in the logical system given by the *Configurator* and, succeding or not, produces a proof object that is given to the *Proof Verifier*. The *Proof Verifier* component checks the proof object and if it is OK, passes it along to the *Proof Optimizer* component. The *Proof Optimizer* tries to find a shorter version of the proof object and gives this version to the *Proof Viewer*, where the proof can be browsed by the user.
>
> The *Configurator* is set according to choices made by the user through the *User Interface*. In order to set other components, the *Configurator* has to access to the *Logics Database* and

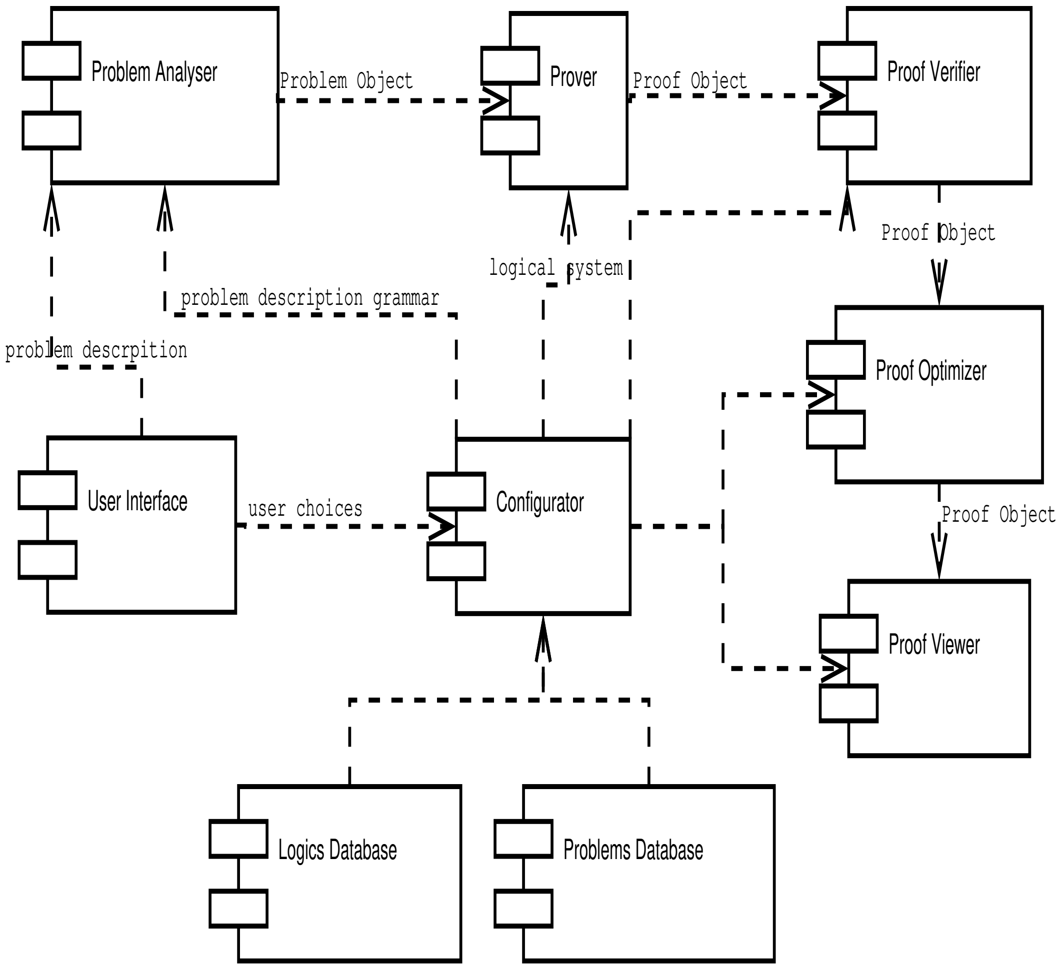

Figure 7: Architecture of the desired ATP system.

the *Problems Database*.

In [73] some representative, broadly-used architectural styles are presented. An architectural style defines a family of systems in terms of a pattern of structural organization. Some of these architectural styles are presented in [12] as patterns for software architecture. Our draft has points in common with two architectural patterns (styles) presented in [12, 73]:

- The *Pipes and Filters* architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allow you to build families of related systems.
- The *Blackboard* architectural pattern is useful for systems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a partial or approximate solution.

It is clear that our draft does not follow these patterns strictly, but that is not the intention of a pattern.

We known that much more can be done in this system, but our objective is to keep this initial description as simple as possible. The main concern of our work is to be able to define strategies for proofs and compare strategies. That is why the *Configurator* component, where the strategies are to be represented and input to the *Prover*, plays a central role in our draft.

6.4 Frameworks

An object-oriented framework is a set of related classes designed for solving a specific problem. Frameworks [41, 24] are well-established as a technology for building object-oriented systems. But frameworks are not usually designed upfront; they evolve. A rule of thumb is that you must have implemented at least three applications in a given subject area in order to derive a framework. In the future we may arrive at a position where we can define an aspect-oriented framework for tableau provers.

7 Conclusion

In this report we have given an account of the current state of our project of implementing tableau provers with object-oriented programming, design patterns, aspect-oriented programming and some techniques from Extreme Programming. We are still in the beginning, but we have just finished an object-oriented implementation of a KE tableau prover.

Our goal is to be able to give a satisfactory modularization of proof strategies for the KE system. We believe that by using aspect-orientation along with object-orientation and other techniques, this modularization will be possible.

References

[1] asato (asato@ncfreak.com). Refactoring in AspectJ, September 2003. http://www.ncfreak.com/asato/doc/refactoring-in-aspectj.html.

[2] Aspect-Oriented Software Association. AOSD Technology, December 2003. http://www.aosd.net/technology/index.php.

[3] Aspect-Oriented Software Association. aosd.net, December 2003. http://www.aosd.net.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Pub Co, Reading, MA, second edition, 2003.

[5] Kent Beck and Ralph Johnson. Patterns generate architectures. *Lecture Notes in Computer Science*, 821:139–149, 1994.

[6] Bernhard Beckert, Richard Bubel, Elmar Habermalz, and Andreas Roth. jTAP - a Tableau Prover in Java. Universitat Karlsruhe, February 1999.

[7] Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[8] K. Broda, M. D'Agostino, and M. Mondadori. A Solution to a Problem of Popper, 1995.

[9] A. Brown and K. Wallnau. The current state of CBSE. *IEEE Software*, 15(5):37–46, September/October 1998.

[10] Bill Burke. JBoss Aspect Oriented Programming, 2003. http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop.

[11] Bill Burke and Adrian Brock. Aspect Oriented Programming and JBoss, May 2003. http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html.

[12] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John-Wiley and Sons, 1996.

[13] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, London, 1973.

[14] P.R. Cohen and E.A. Feigenbaum. *The Handbook of Artificial Intelligence 3*, chapter XII - Automatic Deduction, pages 75–124. William Kaufmann Inc., Los Altos, CA, 1982.

[15] Robert Constable and Cristoph Kreitz. Automated Reasoning, 2002. Slides from a presentation available at http://www.cs.cornell.edu/Info/People/kreitz.

[16] M. D'Agostino and M. Mondadori. The taming of the cut: Classical refutations with analytic cut. *Journal of Logic and Computation*, pages 285–319, 1994.

[17] Marcello D'Agostino. Are Tableaux an Improvement on Truth-Tables? Cut-Free proofs and Bivalence. Avaliable at: http://citeseer.nj.nec.com/140346.html.

[18] Marcello D'Agostino. Tableau methods for classical propositional logic. In Marcello D'Agostino et al., editor, *Handbook of Tableau Methods*, chapter 1, pages 45–123. Kluwer Academic Press, 1999.

[19] José Eduardo Zindel Deboni. *Modelagem orientada a objetos com a UML*. Futura, 2003.

[20] Wagner Dias. *Implementações de Tableaux para Raciocínio por Aproximações*. Master's thesis, Departamento de Ciência da Computação, Instituto de Matemática e Estatística, Universidade de São Paulo, 2002.

[21] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44, 2001.

[22] U. Endriss. A KE based theorem proving assistant, 1996. Master's thesis, Department of Computing, Imperial College, London, 1996.

[23] Luis Fariñas del Cerro, David Fauthoux, Olivier Gasquet, Andreas Herzig, Dominique Longin, and Fabio Massacci. Lotrec: the generic tableau prover for modal and description logics. In *International Joint Conference on Automated Reasoning*, LNCS, page 6. Springer Verlag, 18-23 juin 2001.

[24] Mohamed Fayad and Douglas Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[25] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA)*, 2000. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf.

[26] Robert E. Filman. What is Aspect-Oriented Programming, Revisited. In *Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming*, June 2001.

[27] Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer-Verlag, second edition, 1996.

[28] Melvin Fitting. Introduction. In Marcello D'Agostino et al., editor, *Handbook of Tableau Methods*, chapter 1, pages 1–43. Kluwer Academic Press, 1999.

[29] Martin Fowler. *Analysis Patterns: Reusable Object Models.* Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1997.

[30] Martin Fowler. Is There Such a Thing as Object-Oriented Analysis? *Distributed Computing*, (10):40–41, October 1999.

[31] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Adisson-Wesley Longman, 1999.

[32] Martin Fowler. Who Needs an Architect? *IEEE Software*, (1):2–4, August 2003.

[33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Adisson-Wesley, 1994.

[34] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java.* John Wiley & Sons, 2003.

[35] William Grosso. Aspect-Oriented Programming and AspectJ. *Dr.Dobbs Journal*, August 2002.

[36] Jan Hahnemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, ans Applications (OOPSLA)*, pages 161–173, November 2002. http://www.cs.ubc.ca/˜jan/AOPDS/.

[37] R. Hähnle. Tableaux and related methods. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 3, pages 100–178. Elsevier Science, 2001.

[38] Richard Hightower and Nicholas Lesiecki. *Java Tools for eXtreme Programming - Mastering Open Source Tools including Ant, JUnit and Cactus.* Wiley, 2002.

[39] Scott Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew W. Appel. CUP User's Manual, 1999.

[40] Object Technology International Inc. Eclipse Platform Technical Overview, February 2003.

[41] Ralph E. Johnson. Components, frameworks, patterns. *ACM SIGSOFT Symposium on Software Reusability*, 1997.

[42] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[43] Louis H. Kaufmann. The Robbins Problem: computer proofs and human proofs. *Kybernetes - The International Journal of Systems and Cybernetics*, 30, 2001.

[44] Gregor Kiczales. Interview with Gregor Kiczales, July 2003. http://www.theserverside.com/events/videos/GregorKiczalesText/interview.jsp.

[45] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44:59–65, 2001.

[46] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[47] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[48] Gerwin Klein. JFlex User's Manual, 2001. Available at http://www.jflex.de.

[49] Michael Kohlhase and Carolyn Talcott. Database of existing mechanized reasoning systems, 2003. Available at http://www-formal.stanford.edu/clt/ARS/systems.html.

[50] Ramnivas Laddad. I want my AOP!, Part 1. *Java World*, January 2002.

[51] Ramnivas Laddad. I want my AOP!, Part 2. *Java World*, March 2002.

[52] Ramnivas Laddad. I want my AOP!, Part 3. *Java World*, April 2002.

[53] Ramnivas Laddad. *AspectJ in Action.* Manning, 2003.

[54] Ken Wing Kuen Lee. An Introduction to Aspect-Oriented Programming, August 2002. Reading Assignment. COMP 610E 2002 Spring Software Development of E-Business Applications. The Hong Kong University of Science and Tcehnology.

[55] Nicholas Lesiecki. Test flexibly with AspectJ and mock objects. *IBM's Developer Works*, May 2002.

[56] Karl Liebebrherr, David H. Lorenz, and Johan Ovlinger. aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 542–565, 2003.

[57] Katharina Mehner and Awais Rashid. GEMA: A Generic Model for AOP. In *Belgian and Dutch Workshop on AOP, Twente, The Netherlands*, 2003. http://www.comp.lancs.ac.uk/computing/oop/Publications.php.

[58] Katharina Mehner and Awais Rashid. Towards a Generic Model for AOP (GEMA). Technical Report CSEG/1/03, Computing Department, Lancaster University, 2003. http://www.comp.lancs.ac.uk/computing/oop/Publications.php.

[59] Tzilla Elrad (Moderator), Mehmet Aksit, Gregor Kiczales, Karl Liebherr, and Harold Ossher. Discussing Aspects of AOP. *Communications of the ACM*, 44:33–38, October 2001.

[60] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten. Does Aspect-Oriented Programming Work? *Communications of the ACM*, 44:75–77, 2001.

[61] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis.* Morgan Kaufmann, San Francisco, 1998.

[62] William F. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, Urbana-Champaign, IL, USA, 1992.

[63] Carlos Perez. Refactoring to Aspects, May 2003. http://www.artima.com/weblogs/viewpost.jsp?thread=4842.

[64] Eduardo Kessler Piveta and Luiz Carlos Zancanella. Observer Pattern using Aspect-Oriented Programming. In *Third Latin American Conference on Pattern Languages of Programming*, 2003.

[65] Awais Rashid and Lynne Blair. Editorial: Aspect-oriented Programming and Separation of Crosscutting Concerns. *The Computer Journal*, 46(5):527–528, 2003.

[66] Charles Richter. *Designing Flexible Object-Oriented Systems with UML.* Macmillan Technical Publishing, 1999.

[67] Don Roberts and Ralph Johnson. Evolving Frameworks. *????*, 12(1):23–41, January 1965. Avaliable at http://st-www.cs.uiuc.edu/users/droberts/evolve.html.

[68] J. A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, January 1965. Reprinted in [74].

[69] Start Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995. Preis: 70,- DM.

[70] Satisfiability suggested format, 1993. Avaliable at http://www.satlib.org.

[71] Satisfiability library, 2003. http://www.satlib.org.

[72] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.

[73] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline.* Prentice Hall, 1996.

[74] Jörg Siekmann and Graham Wrightson, editors. *Automation of Reasoning: Classical Papers in Computational Logic 1957–1966*, volume 1. Springer-Verlag, 1983.

[75] Raymond M. Smullyan. *First-Order Logic.* Springer-Verlag, 1968.

[76] Sérgio Soares and Paulo Borba. AspectJ - Programação orientada a aspectos em Java. *Tutorial no SBLP 2002, 6o. Simpósio Brasileiro de Linguagens de Programação. 5 a 7 de Junho, PUC-Rio, Rio de Janeiro, Brasil*, pages 39–55, 2002.

[77] Geoff Sutcliffe and Christian Suttner. The CADE ATP System Competition, 2003. Available at http://www.cs.miami.edu/ tptp/CASC.

[78] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Second edition, 2003.

[79] Tableaux conference, 2003. http://i12www.ira.uka.de/TABLEAUX/.

[80] AspectJ Team. The AspectJ Programming Guide, December 2003. http://dev.eclipse.org/viewcvs/indextech.cgi/ checkout /aspectj-home/doc/progguide/index.html.

[81] Andrei Voronkov. Algorithms, Datastuctures, and Other Issues in Efficient Automated Deduction. In *International Joint Conference on Automated Reasoning*, LNCS, pages 13–28. Springer-Verlag, 18-23 juin 2001.

[82] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications.* Prentice-Hall, 1984.

[83] Michael J. Yuan and Norman Richards. Lightweight Aspect-Oriented Programming - Puuting the Interceptor pattern to work. *Dr.Dobbs Journal*, August 2003.