# Modular Automated Theorem Prover

Tomer Libal (shaolin@logic.at)

22 May 2008

**Abstract**

The program described in this document aims to enable an extensible theorem prover using the resolution method. All the components in the prover, except the core engine, are modular and can be easily re-implemented or extended. For some of the components in the first version of the prover, simple default implementations were created while for others more advanced algorithms were used.

## Contents

# 1 Introduction

The reason for designing another automated theorem prover, despite the variety and expertise of the existing ones, is to have a framework that will enable customization and enhance research in automated deduction by providing the tools to build and test new algorithms. The price, of course, is performance as specialized theorem provers are built for performance in ways that will surpass all attempts to do so in a modular and generic one. On the other hand, the ability to optimize each component separately may enable good performance even on a generic tool.
This document will start with the architecture in chapter 2 and continue with explaining the

different components in chapter 3. Chapter 4 will specify some elements that are still missing. Chapter 5 will give examples of possible extensions and a short explanation of how to extend a module. The last chapter (6) will deal with the tests and performance of the tool.

### 1.1 Interactive commands reference

In this section, the supported commands to the InteractiveRefinement via the PromptUserCommand, will be discussed. The Prover is launched by running the Prover binary (Normally in the Prover/src subdirectory). The Prover then read the input file defined in the configuration (see 2.7) and depending on the mode defined in the configuraion, it may start in interactive mode. In the interactive mode, the user can interact with the Prover using the following commands:

1. tX - where X is a number, which represents the number of steps the prover is to execute without further user intervention. A step is considered to be an attempt to generate resolvents from a pair of clauses (or a sequence in case the deduction engine can handle more than just two clauses, i.e. Hyper Resolution).

2. sX - Searching utility, where X can be any term contained in a clause in the current clause set. This command will return all clauses (together with their indices) that contain the specified term.

3. x y - This command expecets two natural numbers and return the result of trying to resolve the two clauses, whose indices were represented by x and y.

## 2 Architecture

The prover is programmed in C++ with an emphasize on Object Oriented paradigms, an attempt was made to try and use appropriate object oriented aspects in order to solve the architectural constraints.

### 2.1 Components

The core of the prover is the DeductionEngine interface. This part is responsible of looping and querying over the Refinement interface until it is exhausted or an empty resolvent is found. Each sequence of clauses from the Refinement is being parsed by the ResolvingUnit interface in order to produce resolvents. Each of the valid produced resolvents is then being processed by a sequence of ResolventPostProcessor interface implementors which are ordered by priority, a failure to pass a post processor will reject immediately the resolvent. The main.cpp file first reads an XML file that defines the dependencies between the different components similarly to the way it is done in the Java Spring framework, although as there is no reflection in C++, it is done currently in a straight forward way. The outer most component of the prover is the UserCommand interface which executes the program. It has access to the ClausesInput interface which reads the input clauses set, a NodePrint interface which outputs the results and the DedcutionEngine which was already described. Another interface which is available to any component that requires it is a VariableNameGenerator for generating unique names of

variables in variants. There is a special implementation of the Refinement interface, which is the InteractiveRefinement, which is wired with an implementation of the ManagedRefinement interface and has access to the UserCommand in order enable interactive usage of the prover.

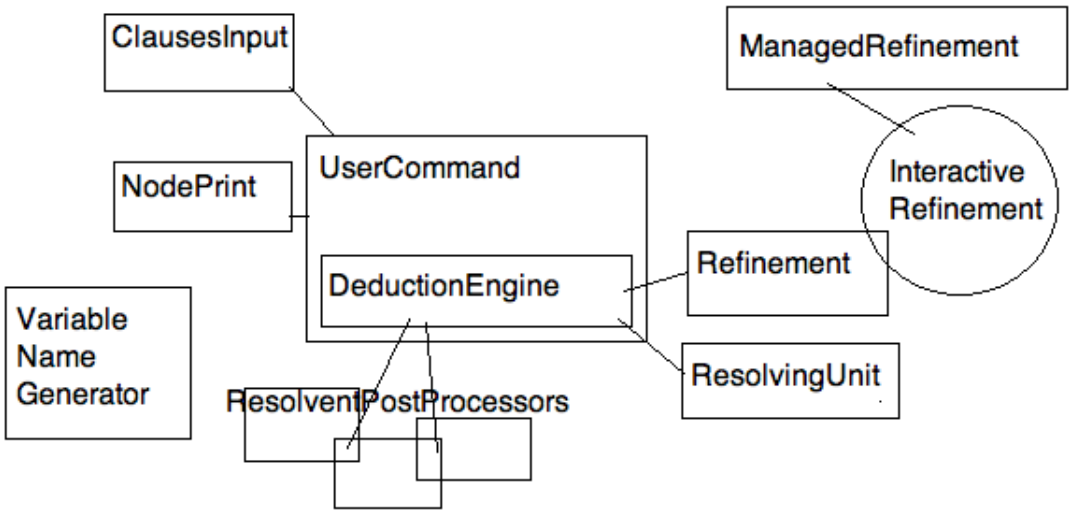


Figure 1: Components Architecture

## 2.2 Proof Objects

Another level in the architecture is the construction of trees of information alongside the process of generating resolvents. These trees represents all the decisions and components in the process and are divided between DataProofNode which extends to Clauses, Resolvents and Unifiers, and InformativeProofNode which extends to ClauseStrategy and ClauseStrategyRuling right now and can be extended to represent more information in the process. Both interfaces extends the ProofNode interface which is the core element which is parsed later by the NodePrint interface to output the Proof itself.

## 2.3 Files Structure

The classes reside in separate files mostly, so all concrete classes will have both a separate header file and source file and some of the interfaces have only header file.
The main library is the DeductionEngine library which holds all interfaces, the DeductionEngine interface and the default and basic implementations. There are three more extensions, each reside in each own folder and library. The ClausesParser is an implementation of the ClausesInput interface. The UnificationResolver is an implementation of the ResolvingUnit interface and the VectorIndexingSubsumption is an implementation of the ResolventPostProcessor which does forward subsumption. The last folder is the main program, the Prover, which has the main file that reads the configuration file and a default implementation to the UserCommand interface.

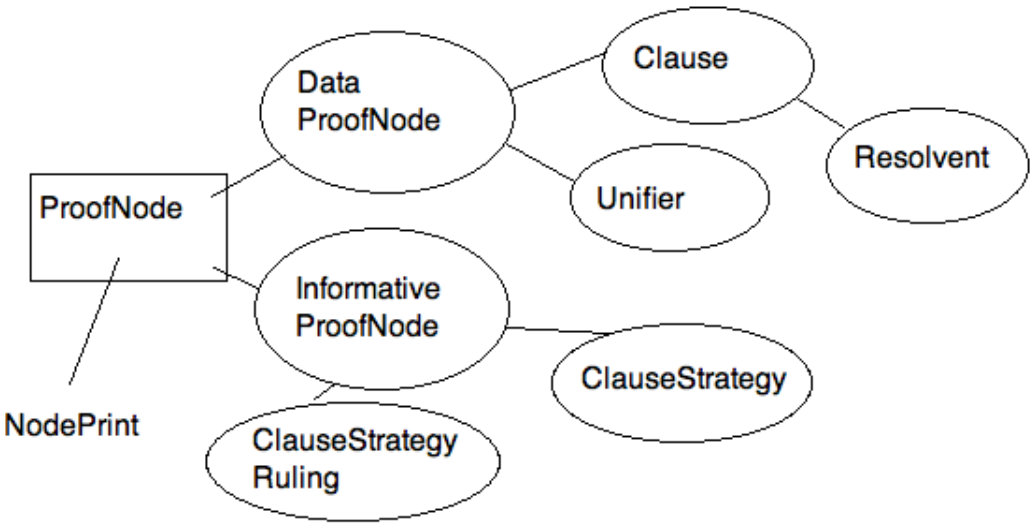


Figure 2: Proof Objects

### 2.4 Exceptions Handling

The main class catches exceptions of type ATPException while there concrete exceptions extends it like RefinementException and ResolvingException.

### 2.5 InputSet Events

The communication between the modules with regard to changes done to the clause set (addition, deletion or even rejection of new clauses) is done via events. The post processors are the only modules which are allowed to change the set (Therefore the SimpleInserter post processor that simply insert a new resolvent that passed all the other processors). Each one of the can fire events via methods implemented in the ResolventPostProcessor interface. The refinements and other modules can listen to the processors and act accordingly, for example, the Subsumption post processor, which handles its own data structure relating to the clauses, also listens to the simple input post processor in order to update its data structure and in case there will be a backward subsumption post processor, the forward (and backward) subsumption post processors will have to listen to a new SimpleDeletePostProcessor in order to delete clauses from their data structures.

### 2.6 The Proof Search

According to [2] the search space is separated into three elements, Refinements, Redundancy tests and Heuristics. The Refinement interface deals with both Refinements and Heuristics as from a practical point of view it seems that as the Refinement manages the data structures that holds the elements, it will also be best suited to decide the order of the elements. For example,

in the UnitRefinement there are 6 data structures, 5 lists of clauses, positive units, negative units, etc. and one queue that holds only units with the current paired element and each time an element is dequeued, we try to insert another one with the next element attached. So the Heuristics here is done according to the storage of elements in the refinement. The heuristic, which is a BFS based on a tree with a branch for every unit clause and each one has all the possible counter clauses as single child at a time, is also contained inside the Refinement. This can easily change in other Refinements by implementing an interface Heuristic for the specific refinement (I cannot see yet how it can be completely general) which can be extended by different Heuristics.
Once the two clauses were selected there is another BFS algorithm, this time in the BinaryDeductionEngine (The only implementation of DeductionEngine so far), which iterates over all the possible combination of literals among every two clauses and pass them (if the ClauseStrategy allows that) to the ResolvingUnit. The resolving unit may use factorization in order to further search for resolvents among the factors (again with the approval of the ClauseStrategy). The ClauseStrategy is being created by the refinement when it decides the next clauses to be passed to the DeductionEngine and so can also affect this part of the search.
The third element in the proof search, the Redundancy, is being controlled by the post processors which are the only parts of the tool which can manipulate the clause set (by using events).

### 2.7 Customization

The program settings can be changed using the setup.xml file. The idea of the file was to mimic the work of the dependency injection as exists in the Spring Framework [5] but the file although in xml format, works more as a property file. As there is no reflection in c++, all the dependancies are hardcoded in the class SetupParser. The setup.xml defines the implementations that should be used, for some implementation more properties need to be set (i.e. for CEResXMLNodePrint an output destination file must be specified). Listed in the next section are all the implementations that can be used each under the name of the abstract type (i.e. the CEResXMLNodePrint should be specified as a property of type NodePrint). The implementation of the SetupParser is very simple and it does not perform some basic tests such as defining two different implementations to the same interface and do not defining a required implementation at all. Keeping true to the c++ legacy, the outcome can be that it will work, or a segmentation fault.

## 3 Components

### 3.1 ProofEnvironment

Each of the packages should implement its evnrionment which it needs to use. An environment contains all the components that can be used. For example, in the UnificationEvironmentExtension there should be the chosen unification algorithm and para modulator. The super environment, which is defined in the Prover package, extends all small refinements and set all the vaules in the SetupParser.

## 3.2 ProofProcess

When a proof starts, a process is created. This process exists as long as we try to prove a clause and it may exist longer than a deduction engine or refinement, which may be replaced during the execution.

## 3.3 ClausesInput

This interface is responsible of generating the initial clauses set that is to be refuted. It has two abstract methods, the getInputSet which returns the InputSet object which contains the initial clauses and the execute command which is called in order to create the set by the specific ClausesInput implementation. There are two implementations to the ClausesInput which are the ClausesDriver and Prover9Driver. I could not find how to execute automake on bison parser files with a name different than y.yab.yy and so both implementations use the same file names and cannot be linked on the same time. if one wants to switch between them he must change both the code in SetupParser and the makefiles. The default current implementation is Prover9Driver.

### 3.3.1 SpiritParser

The inputfile format is as follows:

1. A clause is a list of literals seperated by single or sign (|).
2. A positive literal is a term. A negative one has '-' before it.
3. a term may be a function, constant or variable
4. functions start with varchar and any alphanumeric sequence. immediately after there is a list of subterms enclosed by brackets and seperated by commas.
5. constants start with a,b or c and any alphanumeric sequence.
6. a variable start with varchar and any alphanumeric sequence.
7. varchar is d-z.
8. comments can appear after the comment char '%' (until the end of the line).

### 3.3.2 Prover9Driver

This parser uses input similar to Prover9. constants and functions must be enclosed in quotes while variables are without quotes. negative literals are with '-' before them. comments can be added after the symbol '%'. Currently the name of functions, constants and variables is being parsed the same and is defined by

```
[a-zA-Z0-9\+\*_{}<]+
```

In order to extend this, please add new symbols to the ClausesScanner.ll file and run make again. Comments can appear only after a clause.

## 3.4 ResolvingUnit

The ResolvingUnit interface is being called by the DeductionEngine module in order to create a list of resolvents out of two clauses. In addition this interface must support unification of two terms. Issues of factorizations, unification algorithms, et.c can be solved by creating a generic instance of ResolvingUnit which will accept interfaces for solving these issues. The current implementation does not allow this, as well para modulation, etc., There is only one implementation which is the UnificationResolver.

### 3.4.1 UnificationResolver

This implementation, which resides in the UnificationResolver folder is using a Factorizer interface in order to compute the factors, an UnificationAlgorithm to do unification and ParaModulator for the para modulations.

### 3.4.2 Factorizer

This interface receives a clause and computes all its factors (including the clause itself).

### 3.4.3 NoFactorizer

A stub that returns only the input clause so no factorization is being done

### 3.4.4 ResolvedLiteralFactorizer

This Factorizer focuses on the literals that are going to be resolved upon as defined in [2], as follows: First we do unification on the two literals chosen, in case the unification succeeds, we retrieve all factors of each clause with the generated unifier such that the deleted term is unifiable with the resolved upon literal. We then combine all possible of pairs out of these factors and return them.

### 3.4.5 UnificationAlgorithm

There are several modules which uses unification and each can use another instance, right now the setup parser bind all of the modules (para modulator, unification resolver and subsumption post processor) to the same instance. This interface returns a unifier for two clauses for specific literals

### 3.4.6 SimpleUnificationAlgorithm

A simple implementation that does a straightforward simple unification by recursive descent as defined in [1] plus the occur check

### 3.4.7 NoParaModulator

This option disables para modulation

### 3.4.8 ParaModulator

This interface returns a set of para-modulants for a given pair of clauses and specific indices, it has currently two implementations.

### 3.4.9 SimpleParaModulator

This implementation returns the first para-modulant it finds.

### 3.4.10 BFSParaModulator

This implementation returns all the para-modulants it finds.

## 3.5 Refinement

The Refinement interface is responsible of ordering and combining the clauses given to the deduction engine. The interface support giving a sequence of clauses, in which case the resolvent of each two clauses will be resolved with the next clause in the sequence. The Refinement interface should implement the method that returns the sequence. In addition the Refinement extends the InputSetEventListener which listens to changes done to the clause set (Addition, deletion and rejection of clauses), so it can update its inner data structures.

### 3.5.1 InteractiveRefinement

InteractiveRefinement is an implementation of Refinement which is a wrapper around another Refinement, the ManagedRefinement. The main addition of this wrapper is to add communication with the UserCommand in order to get input and send output to the user with regard to the action available and the status of the InputSet. The UserCommand is the same as is defined in the settings file.

### 3.5.2 ManagedRefinement

This interface extends the Refinement interface and adds methods that enable it to be supported by the InteractiveRefinement, so only instances of this interface can be wrapped by the interactiveRefinement. These instances are also regarded as normal Refinements and can be used without the wrapper as well (in a completely automatic mode).

### 3.5.3 UnitRefinement

The first concrete Refinement is the UnitRefinement which extends the ManagedRefinement interface so it can be used both in the interactive mode and in automatic mode. The UnitRefinement works on two clauses at a time, one of which must be a unit clause. In order to achieve this the refinement manages 6 data structures, the first five hold the clauses divided according to type and polarity, unit negative and positive, non-unit negative and positive and mixed clauses and the last data structure is a queue of unit elements, each with the last clause it was paired with from the other 5 queues (positive units are paired with negative units, negative non-unit and mixed and so for negative units). Each call to return a new sequence of clauses to

the deduction engine pops the next unit, fits to it the corresponding clause, computes the next index and re-inserts it into the queue.

#### 3.5.4 PositiveRefinement

Another Refinement which extends the ManagedRefinement is the PositiveRefinement. It is a simple implementation which ensure one clause at least is positive and has an history list to make sure it does not parse combinations that where chosen by the user interactively.

### 3.6 ClauseStrategy

The deduction engine and the resolving unit consult with the clause strategy in order to decide if to try to unify a specific literal in the search for resolvents or create a specific factor after unifying. This is done by generating a ClauseStrategyRuling class for every query which describes the decision made. So far only default instances of these interfaces were used in the UnitRefinement and so, no information is stored in this objects. In genera these objects are part of the proof tree and can be printed out using the NodePrint interface.

### 3.7 DeductionEngine

It is being activated by the UserCommand with an initial set of clauses and is running a loop that retrieves sequences of clauses from the refinement until an empty clause is found or the refinement is exhausting the clauses. For each sequence the engine goes over the element and computes the resolvents of each clause with the last resolvent. It also retrieves the sequence strategy which is used to decide on which literals to try unification and is created by the Refinement when returning the clauses sequence. The computation of the resolvents is done by trying out all combinations of literals in the two clauses which are approved by the strategy. Once two literals are chosen, the two clauses are passed to the ResolvingUnit in order to produce the resolvents. The Deduction engine also calculates the processing time so it can cancel the execution when a timeout is reached (if it is defined). Once a resolvent is generated it is being passed through the post processors which have fixed order, if a post processor rejects the resolvent, it is not being passed to the next post processor.

A note - There is currently only one implementation called BinaryDeductionEngine which throws exception when being feed with Resolving Sequences with more or less than 2 clauses, so it does not support hyper resolution for example right now. There can be many extensions to this class like a SequenceDeductionEngine that support longer sequences and MultiThreaded-DeductionEngine that support multithreading in parsing clauses.

#### 3.7.1 BinaryDeductionEngine

Currently this is the only implementation. It is single threaded binary resolution engine.

### 3.7.2 Clause

The Clause object, which represents, naturally, a clause, has two vectors of literals, for positive and negative. It also contains the ClauseContext object, which holds all variable mappings and plays an important role in the prover as the literals of the clause must be passed through the context in order to bind all variables. The Unifier class also extends the ClauseContext and represents a unification mapping. In case there is a need to extend the clause class for use in specific refinements (such as labeling the clauses), there is a ClauseInfo class inside Clause which can be used or extended.

## 3.8 NodePrint

The NodePrint interface is being used to output the empty sequent proof tree in case there is one or output the proof trees of all generated resolvents. It is an implementation of the Visitor design pattern. The NodePrint is called recursively on all nodes of the Proof tree, starting with the final resolvent and for each node, Resolvents, Unifers, Strategies, Clauses, etc. the specific method is called in the interface in order to output the result.

### 3.8.1 LatexNodePrint

This implementation produces latex code from the proof tree, it outputs the resolvents, clauses and unifiers only in latex format. It takes a filename as the target latex file.

### 3.8.2 CEResXMLNodePrint

This is currently the default implementation. it produces XML code which can be read by ProofTool or the CERes project. Currently only the following is being displayed: res steps and para modulation steps. The rest, like the substitutions, can easily be added in the code (there are empty methods ready).

## 3.9 VariableNameGenerator

This interface is being used in order to generate new variable names to be used in variants.

### 3.9.1 SimpleVariableNameGenerator

This simple implementation just increases a counter and appends the current count to the variable name.

## 3.10 ResolventPostProcessor

The post processors are responsible for managing the new resolvents, each is given a priority defined in the setup file and which defines the order on which it processes the resolvent with regard to the other processors. The post processor has methods for firing InputSetEvents in order to tell the refinement and other listeners (for example some post processors are themselves listeners so they can update their own data structures on removal or addition of clauses). The post processor must implement the postProcess method, which is given the new resolvent.

### 3.10.1 SimpleInsertPostProcessor

This simple post processor is simply adding the resolvent to the input set and then firing an event that the resolvent was added.

### 3.10.2 TautologyEliminationPostProcessor

This post processor checks if the resolvent is a tautology, if it is it fires a rejection event. The test is done by comparing positive literals to negative ones.

### 3.10.3 ForwardSubsumptionPostProcessor

This post processor checks for forward subsumption. It uses Stillman algorithm as defined in [2] in order to check if one clause is subsumed by another one. It does not use so far the more sophisticated algorithms defined in [2] but is using an interface for an algorithm so another algorithm can be plugged in. Another optimization is the usage of vector indexing in order to find the clauses which may subsume the new resolvent as defined in [4]

## 3.11 UserCommand

The interfacing to the user is done via the UserCommand interface. This is the interface that the main file interact with in order to initiate the prover and it is also the interface given to the InteractiveRefinement in order to interact with the user. The UserCommand can be extended to support graphical user interfaces or to serve as an adapter between the program and an external user interface (or s web server, etc.),

### 3.11.1 PromptUserCommand

This is the default implementation which enables interaction via the shell.

## 3.12 SetupParser and the main file

The idea here was to use a lightweight container in order to manage the life cycle of the objects and wire them together as is done in the Spring framework [5]. But there is no such support in C++ as there is no reflection on objects and classes. The closest library found was PocoCapsule [3] but it is supporting currently only a limited number of operating systems. The solution for now was to create an XML like property file which lists the interfaces and their implementations (the setup.xml file). The implementations are then hardcoded in the SetupParser and are instantiated when chosen from the file. Any new implementation is required to be added to the SetupParser as well. The order of the listings in the "XML" properties file is also important, for instantiations which requires another component in the constructor, they should be declared after and for those setting it later, should be declared later. The setupParser set all the components into the environment class. The SetupParser generates at the end a userCommand instance loaded with the environment and the main file then call the execute method and catch all exceptions.

# 4 Unimplemented parts

Currently there is only one part which is not implemented and is being required by some refinements (which are also not implemented so far). The ResolvingSequence class supports a list of clauses to be fed up to the engine as required by refinements such as Hyper Resolution, but right now any sequence with length different than 2 will be rejected by the engine.
Furthermore, there are several parts which have only minimal implementation, for example the interaction between UserCommand and the InteractiveRefinement is done via a minimal interface that supports the current refinements needs for interaction with the user (i.e. asks for the user to choose two clauses by indices and presents the possible choices to the user). In case a more extensive interface will be required then it should be added to both the InteractiveRefinement and the UserCommand interface in a generic way and then also an update to all UserCommand concrete implementations should be done.
Another minimal implementation is the ClauseInfo object that is attached to each Clause, the info object should enable more parameters to be attached to each clause (like labeling) but as there is no current need for any, the class was left empty. The implementation I thought about is by adding a map to hold properties.

# 5 Extensions

## 5.1 Required Extensions

Here is a list of Extensions that are required in order for the prover to be "complete":

- Better refinements, right now only the incomplete UnitRefinement was created.

## 5.2 Optional Extensions

- Improving the Forward Subsumption PostProcessor to support more efficient algorithms.
- Improving the paramodulation algorithm.
- Adding a new deduction engine which supports multithreading and sequences of clauses (for hyper resolution).
- Implementing a better UI, maybe a GUI on top of the UserCommand.
- Extending the output of the CEResXMLNodePrint to cover also the unifiers.
- Adding more Post Processors (like backward subsumption)
- Creating a composite Refinement which uses several other Refinements according to the type of clauses (To be closer to the way efficient theorem provers are working).

### 5.3 Extending the interfaces

In order to extend the interfaces, one must implement on of the interfaces of course and then add an entry for it in the SetupParser in order for it to be instantiated. Then the new name should appear in the setup.xml file next to the interface it has implemented. As there is no reflection, the Prover module will have to be recompiled after the change to SetupParser was done and its Makefile.am will have to include the new library.

## 6 Tests

## References


[1] Franz Baader and Wayne Snyder. *Unification Theory, Handbook of Automated Reasoning, Chapter 1.* Elsevier Science Publishers, 1999.

[2] Alexander Leitsch. *The Resolution Calculus.* Springer, 1997.

[3] Pococapsule - pocomatic software.

[4] Stephan Schulz. Simple and efficient clause subsumption with feature vector indexing. *Theoretical Computer Science*, 2004.

[5] Spring framework.