The Design and Implementation of an Object-Oriented Prolog Engine

Giulio Piancastelli*, Alex Benini, Andrea Omicini, and Alessandro Ricci

DEIS, Alma Mater Studiorum, Università di Bologna
via Venezia 52, 47037 Cesena (FC), Italy

Abstract. The implementation of Prolog systems has a long history, starting from the first interpreter written in 1972, to the first Prolog compiler simplifying basic operations like unification and backtracking with the aim of optimizing performances, and to the Warren Abstract Machine, later becoming a de facto standard model. The tuProlog engine differs from those systems in that maximum efficiency is not amongst its primary concerns. Instead, it aims at becoming the enabling technology for building complex systems like dynamic Internet infrastructures based on intelligent components, a domain where logic languages could effectively face, in principle, the key issues of intelligence and interaction. To accomplish this task, a logic engine must meet engineering properties such as deployability, configurability and scalability; to be usable as a development tool, it also must sustain a certain degree of traceability, by allowing the inspection of its internal processes at runtime, thus easing the typical pain of debugging activities. The design and implementation of such a system must follow sound principles: object-oriented code structuring, reuse of established community knowledge under the form of patterns, loose coupling of composing elements, modularity and a clear and clean separation of concerns. In this paper, we present the architecture of tuProlog, based upon a set of managers handling control of sensible parts of the system and operating around a minimal core shaped as a Finite State Machine; then, the representation of Prolog data and the unification algorithm’s implementation are discussed; finally, the resolution part of the engine is illustrated.

Keywords: Prolog, Finite State Machine, object-oriented design.

1 Introduction

The implementation of Prolog systems has a long history [1, 2] starting with the first interpreter written in Algol-W by Philippe Roussel in Marseille in 1972, improved and rewritten in Fortran by Gérard Battani and Henry Meloni an year later. The reasonable performance of that system helped convincing researchers of the usefulness and viability of Prolog as a logic programming language. Consequently, in 1977, David Warren together with Fernando Pereira and Luis Pereira developed the DEC-10 Prolog [3], the first Prolog compiler.

* Contact author. Email: giulio.piancastelli@unibo.it. Phone: +39 0547 634744. Fax: +39 0547 339219.
The main purpose of compilation, in the case of the Prolog programming language, is to simplify each occurrence of its basic operations, i.e. unification and backtracking, by extracting as much information as possible before running the program, in order to reduce its execution time. For instance, the compilation step splits up the unification process in a sequence of elementary operations which many optimizations can be performed within, determines particular situations which may require ad hoc solutions (e.g. tail recursion) and translates the code trying to delay as much as possible the execution of certain operations which could result as useless because of the failure of some other subsequent operations.

As an outcome of his own explorations, by 1983 Warren had also developed an abstract machine, composed by a high-level instruction set and a memory model, which has become the de facto standard implementation technique with the name of Warren Abstract Machine [4]. Along the lines of the WAM, many different variations has been proposed, from the VAM (Vienna Abstract Machine) models [5] to logic engines based on the continuation passing style model of Prolog [6], but the simplification principle underlying compilation still continues to hold, and the quest for optimization seems to be never ending.

The tuProlog\(^1\) engine [7] is instead a different beast. Its main concern is to become a key technology in domains where intelligence and interaction are regarded as the core issues. While the vocation of logic languages has always been to enable the development of intelligent components, they also proved to be effective both as communication and coordination languages, suggesting that they could play a fundamental role in building today’s complex systems like dynamic Internet infrastructures [8]. To accomplish this task, a logic engine must meet some engineering properties such as configurability and scalability; to spread ubiquitously and become as simply deployable as possible, its target platform has to be carefully and properly chosen; and, to be usable as a development tool, traceability is also considered an important feature, in order to enable the inspection of the engine’s internal processes at runtime, thus easing the pain of debugging activities, so typical for logic declarative languages as dense as Prolog. But performance is not a primary concern for that project: in fact, it aims at achieving reasonable rather than maximal efficiency. Every set of requirements dictates, or at least influences, the architecture and the characteristics of the system designed to satisfy it; starting from the choice of Java as its implementation platform, tuProlog is no exception. As a consequence, to become the sound enabling technology for logic components that dynamic, configurable, intelligent and interoperable systems are based upon, a similar soundness is needed in its design and implementation: object-oriented code structuring, reuse of established community knowledge in the form of design patterns, loose coupling of composing elements, modularity and a clear and clean separation of concerns are the guiding principles for the construction of such a basic brick. As far as this kind of approach is concerned, we are aware of one precedent only [9], set by the description of an object-oriented Prolog inference engine written in Smalltalk for Gödel, the Graphically Oriented Development Environment for Logic programming.

\(^1\) Available at \url{http://tuprolog.alice.unibo.it}
In this paper, we aim at discussing the design and implementation of the tuProlog engine, starting from its modular architecture composed by a set of managers handling control of sensible parts of the system and operating around a minimal core shaped as a Finite State Machine (FSM). Then, the representation of Prolog data and the unification algorithm, distributed along the data classes in truly object-oriented fashion, are illustrated. Finally, the resolution part of the engine with its various elements is presented, and conclusions are drawn.

2 The Architecture of tuProlog

Two main requirements dictated the initial sketch of tuProlog’s architecture: a minimal Prolog virtual machine, available as a self-contained object, featuring a simple interface; and the need for extendibility of such core by means of dynamically linkable and dischargeable libraries of predicates. To obtain an appropriate degree of modularity, the main elements distinguished by those prerequisites have been broken into smaller components with a single and well determined responsibility. On that account, the system is composed by a set of manager entities which control relevant parts of the Prolog virtual machine, working under the cover of a single object playing the role of a unified, high-level interface to the whole engine: such a design is obtained by applying the Façade [10] pattern. The division into managers, for the inferential core, theories, libraries and primitive predicates, has been carried out to allow each subsystems to grow as independently as possible from the other. and separate the concerns about the sort of extensibility tuProlog does and shall supports by means of libraries from the rest of the engine. However, the user should be shielded by the complexity of those subsystems, both at present state and as they evolve, and presented with an unified interface providing a simple view of the whole Prolog system. The Façade design pattern is employed also with the aim of reducing the number of objects that clients have to deal with and making the system easier to use.

As independent the managers as we would like them to be, they need to communicate anyway, because they have to interact in order to execute the job of a Prolog engine. Yet it is possible to break dependencies between those managers, thus promoting loose coupling by keeping them from explicitly referring to each other. The Mediator design pattern [10] does exactly this: it defines an object that encapsulates how a set of objects interact, and keeps open the chance of varying their interaction independently. So, instead of knowing each other and communicating through direct references, a manager knows the mediator only, and asks him for services which are implemented by collaboration amongst other managers. As depicted in Fig. 1, while the Prolog class plays the role of façade to the system, it is the ManagerMediator class to play the role of mediator amongst the various managers: LibraryManager takes care of tuProlog libraries; PrimitiveManager controls the execution of primitive predicates; TheoryManager administers logic theories; finally, EngineManager governs the running of the Finite State Machine representing tuProlog’s inferential core.
2.1 Library Management

The `LibraryManager` class is responsible for managing the basic configurability means of `tuProlog`, that is libraries of predicates and functors to be dynamically loaded or unloaded even at runtime, while the demonstration activity is taking place into the engine. Since `tuProlog` is by design choice a minimal, purely inferential core, the only *built-in* predicates defined in the engine are: the ones which directly influence the resolution process, such as `cut/0`; the ones which are too basic to be defined elsewhere, such as `fail/0`; and the ones which needs to be defined near the core for efficiency reasons, such as `','/2`. Apart from these very few exceptions, there are no technically built-in predicates, intended as predicates defined statically inside the core: every predicate used by `tuProlog` is defined by a library, and can therefore be defined as “built-in” only in the sense that, and only as long as, the library defining it is loaded into the engine. Unlike most Prolog system, this peculiarity is also shared by those provided predicates and evaluable functors which are defined in the ISO standard. So, in `tuProlog` parlance, *built-in* predicates are those predicates which can never be unloaded from the engine: of course, basic predicates to load and unload libraries such as `load_library/1` and `unload_library/1` are included in this category.

To augment the bare-bones inferential core with functionalities supported by every usable Prolog system, four libraries are provided which get loaded by default at engine’s creation time when the no-arguments constructor of the `Prolog` class is invoked. Those libraries, derived by the base `Library` abstract class which provides common services, are:

- **BasicLibrary**, which defines some basic predicates and functors usually found in Prolog systems, with the exception of I/O predicates;
- **IOLibrary**, which provides some of the standard Prolog I/O predicates, such as `write/1, read/1` and `nl/0`, further separated by other functionalities, so as to enforce orthogonality between interaction and computation;
- **ISOLibrary**, which defines standard ISO predicates not defined in the previous libraries and not concerning the I/O realm;
- **JavaLibrary**, which defines all the predicates for Prolog/Java interaction.
Although tuProlog libraries are expressed in Java, they are not required to be fully implemented in this language. Java-only libraries are just the simplest case, resulting in deterministic libraries since the Java programming language does not inherently supports non-determinism. This is not to be seen as a limitation, but rather as a specific design choice to prevent an undesired jumble of the two paradigms [8]. In fact, whenever needed, non-deterministic libraries, whose predicates and evaluable functors can provide multiple solutions, can also be defined: the basic idea is to add a non-deterministic Prolog layer on top of the deterministic Java layer. To make such a hybrid possible, a Prolog theory can be embedded into a Java string within the library, returned by the public getTheory method, inherited by the base Library class. When loading a library, the LibraryManager class always calls such method, adding the provided theory to the engine’s configuration. In tuProlog parlance, Prolog library predicates are known as library predicates, while Java library predicates are called primitive predicates, or simply primitives. Prolog predicates coming from a program fed by a user to the engine are instead known as user-defined predicates.

2.2 Primitive Management

When a Prolog theory is read by the engine, be it from a library or a user-supplied program, the predicates and functors it contains must be identified as belonging to one of two categories: library or user-defined predicates on the one side, primitive predicates on the other side. The difference between the two sorts lies in the way they are executed during the resolution process. Those predicates being classified in the first category are taken care of by the inferential core and its Prolog execution algorithm; those lying in the second category are not to be found during the exploration, performed by the execution algorithm, of the logic theory contained in the engine, but they need to be somehow invoked as Java methods whenever a subgoal in the resolvent references one of them.

The PrimitiveManager class deals with such identification activity. Each library is able to provide a reference to its methods defining primitive predicates and evaluable functors via Java reflection mechanisms; instances of the PrimitiveInfo helper class are used to keep that reference. The manager class first collects those data; then, for each predicate contained in the body of each clause in the knowledge base accumulated in the engine, its identification service is called by the TheoryManager class. The Term representing the predicate is obtained and, if the predicate belongs to the primitive set, a PrimitiveInfo instance is linked to it. When the time comes that that predicate has to be executed, the PrimitiveInfo instance associated with it takes care of running the primitive by invoking the Java method corresponding to it, exploiting again the Java reflection facility.

2.3 Theory Management

When a tuProlog engine is asked to consult a Prolog program, the theory manager is invoked. The first step consists in running the text in the program through
the parser, which recognizes the syntax of correct sentences in the Prolog pro-
gramming language and creates structures corresponding to the identified terms.
Those structures are instances of classes in the hierarchy representing Prolog
linguistic elements, described in Sect. 4, and are part of tuProlog’s Application
Programming Interface that developers exploit when working with the engine
from the Java side. Interleaved with the parser, the TheoryManager class carries
out its duties: creating the clause database to form the knowledge base of the
engine, and recognizing directives to be immediately executed.

Information contained in the terms hierarchy is not enough for the engine to
perform the execution of a Prolog query. So, a new helper class is used, named
ClauseInfo, which holds a reference to the term object representing a Prolog
clause, and supplies additional data, e.g. a flag indicating if that clause is dy-
namic (hence eligible to be asserted or retracted) or the library’s name which
the clause comes from, in case the clause has been read from the Prolog part
of a tuProlog library. Such ClauseInfo instances are stored in lists of clauses
belonging to the same family, i.e. featuring a head with the same functor and
same arity, which are in turn stored into a hash table. The list guarantees the
reading order is preserved when accessing clauses during the demonstration pro-
cess; the grouping under the same key, constructed from commonalities in the
clause’s head, ensures fast access to members of the same family.

For optimization purposes, the body of each clause is immediately decom-
posed in subgoals as soon as the clause itself is recognized by the parser and the
corresponding ClauseInfo instance is created: by early decomposition at con-
struction time rather than execution time, a possible overhead for the extraction
of subgoals during the resolution process is avoided. The decomposed body of a
clause is stored in a tree structure where each subgoal is represented by a leaf,
and visits are conducted depth-first. Such a structure makes it easier to deal
with parenthesized subgoal conjunctions in a clause’s body, and especially with
second order logic elements in the language. For example, when a subgoal is a
variable, the different subtrees it could reference during subsequent executions
can be comfortably attached to and detached from a tree structure rather than
the list structure other systems usually resort to.

As soon as each clause is recognized from the Prolog text, it is also needed
to check whether it represents a valid directive; if the check is positive, the
directive has to be scheduled for execution with the appropriate timing dictated
by the ISO Prolog standard [11]. Hence, the TheoryManager holds directive
names which the first term in the body of a headless clause is compared to:
when a match is found, a suitable action is dispatched.

2.4 Engine Management

That part of the public interface of the Prolog class that contains operations
dealing directly with the invocation of routines in the inferential core is admin-
istered by the EngineManager. The aim of this class is to loose the coupling
between the façade of the system and the machine realizing the core engine: so,
the functionality in methods like \texttt{solve}, \texttt{solveNext}, \texttt{hasOpenAlternatives} and such are delegated to corresponding methods in the \texttt{EngineManager}'s interface.

Apart from defining some utility methods for the Prolog Finite State Machine implementing the inferential core, the \texttt{EngineManager} class is also responsible for the mechanism that allows solving processes to be nested. As an example, suppose that a goal is being resolved, and that during the demonstration the engine is asked to load a library by means of the \texttt{load_library/1} predicate. That library could contain a Prolog theory, and that theory could include a \texttt{solve/1} directive, to be executed as soon as the Prolog text in the library has been decomposed into \texttt{ClauseInfo} objects and loaded into the engine's knowledge base. This solving process is not executed by the same inferential core instance carrying out the solving process which the new process has been started within: instead, a fresh Prolog virtual machine is created and assigned the new query. Instances of the finite state machine are traced in a data structure with stack semantics, and conveniently pushed and popped when needed.

3 The Prolog Finite State Machine

The inferential core of \texttt{tuProlog} is realized in the form of a Finite State Machine, whose basic architecture is depicted in Fig. 2. The machine is composed by an initial state, assumed as the solving procedure starts by means of a call to the \texttt{solve} method in the \texttt{Prolog} class; four main states representing the activities performed during the resolution process; and four final states which identify the different ways a demonstration may be ended.

![Fig. 2. The Finite State Machine forming the tuProlog's inferential core.](image-url)
3.1 Execution Model

The *Init* state is the main entry point of the FSM when starting a demonstration process and, as its name declares, initializes the core by extracting the subgoals to evaluate from the query and setting up a data structure which represents the *execution context* for the current subgoal. That subgoal is chosen by the next state, which the *Init* state immediately passes control to.

The *Goal Selection* state fetches the next goal to be executed from the sub-goal list. If such a goal does not exist because the resolvent is empty and the demonstration process has thus ended, the existence of a choice point is checked for: if it does exist, the machine ends up in the *TRUE_CP* state, where a next solution to the initial query can be asked; else, the machine shifts to the *TRUE* state and the resolution process ends.

If the resolvent is not empty and the next goal is successfully extracted, its capability of being executed must be verified: it cannot represent a number, otherwise the demonstration fails by switching the inferential machine to the *FALSE* state; and if it represents a variable, special operations are needed to allow this kind of second order logic behavior. The FSM then changes its state to take care of the next step in its execution algorithm: goal evaluation.

Quite predictably, the *Goal Evaluation* state deals with evaluating a single subgoal that the previous state has extracted from the resolvent. If the main functor of such subgoal is bound to a primitive predicate, its execution is triggered, and the machine gets shifted to the *Goal Selection* or the *Backtrack* state depending on the success or failure of the predicate’s evaluation. If, on the other hand, that main functor does not represent a primitive, its evaluation can only be performed by browsing the logic theory contained in the engine, and select a compatible clause, its head unifying with the subgoal. These actions are performed in the *Rule Selection* state. Whenever a Java uncaught *Exception* is thrown during the evaluation of a goal, the machine stops the computation in the *HALT* state, and the stack trace gets printed to the standard error stream.

When entering the *Rule Selection* state from the *Goal Evaluation* state during normal execution, the first thing to do is to gain a set of rules compatible with the current subgoal from the logic theory stored in the engine. If that set is empty, then a backtrack is triggered by setting the machine’s next state to *Backtrack*. When entering the *Rule Selection* state from the *Goal Evaluation* state during a backtracking trail, the set of compatible clauses is supposed to contain at least one element, so the previous check on its emptiness can be avoided.

So, having found a compatible rule with the current subgoal, its evaluation is prepared: a new *execution context* is created; the subgoal is unified with the clause’s head; and the clause’s body is added to the resolvent substituting the subgoal just unified. Then, if the set of compatible rules has open alternatives and the state has not been entered by the *Backtrack* state, a new context for this choice point is created; if there are no alternatives and the rule selection has been triggered from a previous backtrack, where such a choice point context already existed, that context is destroyed. Finally, tail recursion optimization
on execution contexts is performed, and the next state for the inferential core machine is set to \textit{Goal Selection}.

Once entered the \textit{Backtrack} state, a check on the set of compatible clauses for the subgoal corresponding to the latest opened choice point is performed, followed by a transition to the \textit{FALSE} state to immediately make the demonstration fail if that set is empty. Variables are then deunified, and that subgoal is reloaded into the resolvent. Finally, the coherence of the trail of execution contexts is restored by bringing them to the previous state in the demonstration, and a transition is prepared to the next state, namely \textit{Goal Evaluation}, to let the resolution process restart.

3.2 Implementation

The actual implementation of the finite state machine is realized using the State design pattern [10], illustrated in Fig. 3: the pattern is typically used to allow an object to alter its behavior when its internal state changes, by modeling states as objects which encapsulate the different behaviors. In our \textsc{tuProlog} system’s case, it is the inferential core \texttt{Engine}, started by the \texttt{EngineManager} class calling its \texttt{run} method, to play the role of the object whose state changes during the course of a computation, and whose state is therefore modeled as a set of objects with their own behavior. The class hierarchy representing the core’s states is based in the abstract \texttt{State} class, providing the abstract \texttt{doJob} method which contains the state’s behavior in the concrete subclasses. From \texttt{State} all other classes representing the core’s states are derived: \texttt{StateInit}, \texttt{StateRuleSelection}, \texttt{StateBacktrack} and so on, with a clean name pattern matching state names in the FSM.\footnote{Since end states do not feature substantial behavior, they are modeled as a single \texttt{StateEnd} class, whose constructor takes an integer argument identifying the actual state represented by an instance of that class.} At a finer detail, the pattern is implemented by having the \texttt{Engine} class remembering the next state it must shift to, and having state classes directly manage the transitions between each other.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{state_design_pattern.png}
\caption{The State design pattern as implemented by classes in \textsc{tuProlog}’s FSM core.}
\end{figure}
3.3 Benefits

The architecture of the inferential core, modeled as a Finite State Machine, leads to some advantages from the point of view of non-functional properties generally required by most systems. Concerns in tuProlog’s internals are clearly separated: this opens possibilities for both extensions of the machine with concepts that fit the current model and configuration of the machine’s behavior at the single state grain. Exceptions in the Prolog language are an interesting example of potential extension for the current core engine: tuProlog still does not support them, but their insertion in the execution flow could be modeled as another state in the machine. Called, for instance, *Exception*, this new state would be assumed by the FSM whenever, in *Goal Evaluation*, a Prolog exception is thrown; the state’s *doJob* method then would need to find the appropriate clause describing the catching of the exception and prepare the handler for execution by making the machine switch to its *Goal Selection* state, or trigger a backtrack by moving the machine to the *Backtrack state* just in case the exception is not to be managed.

Separation of concerns offers the possibility for configuration options in the behavior of each modeled state abstraction. In particular, the *Backtrack*, *Rule Selection* and *Goal Selection* states would be interested: so, the property of configurability would be applied not only to the whole engine, by means of libraries, but even to the inferential core, by some mechanisms allowing redefinition of backtrack, search and selection rules.

Finally, the nature of the Finite State Machine could also be exploited for debugging purposes. Transitions between states could be controlled from an external entity in a semi-automatic way, so as to realize a transparent engine whose inferential core’s behavior might be inspected during the execution of queries.

4 Term Representation

The linguistic elements of the Prolog language have to be mapped onto a hierarchy of Java classes forming a part of the fundamental Application Programming Interface which tuProlog developers use to manipulate the engine’s components. Consequently, such a mapping of untyped entities in a declarative world onto typed entities in an object-oriented world needs to be as direct and simple as possible, while preserving all the expressiveness of the Prolog language. As far as the design of that hierarchy is concerned, patterns come handy again: the Prolog terms taxonomy has indeed been structured using the Composite design pattern [10], by following the idea of composing objects into tree structures to represent part-whole hierarchies, in order to let clients uniformly treat individual objects and compositions of objects. A typical use for the Composite design pattern is the building of the abstract syntax tree representing internal grammar data structures in an interpreter.

At the top of such hierarchy lies the *Term* abstract class, playing the *component* role and providing common services and operations to the whole taxonomy.
Derived from Term, next comes the Struct concrete class, which plays the role of composite and is one of the most important classes of the hierarchy, since it represents most of the elements in the language: atoms, compound terms including clauses (with functor ‘:-’ and arity 2) and conjunctions (with functor ‘,’ and arity 2), lists. Special mappings distinguishing the empty list from other atoms, or lists from other structures, to allow for a more efficient implementation, are sacrificed in favor of uniformity of representation at the API level. The other fundamental element in the terms hierarchy is the Var class, which represents Prolog variables and plays the role of leaf in the Composite design pattern. Finally, numbers are dealt with by a sub-hierarchy of leaf elements representing integers and floating point numbers with as fine a granularity as the one employed by the Java language, used to implement the tuProlog engine. The terms taxonomy is illustrated in Fig. 4

![Fig. 4. The Composite hierarchy representing Prolog terms.](image-url)

One of the conceptual advantages of working on top of an existing virtual machine is that it lets developers disregard the lower level details of the implementation of their abstractions. Since raw performance is not one of the primary objectives of the tuProlog project, there is no need to address problems such as memory representation of elements at word or bit granularity, cell tagging, efficient value checking, comparison and assignment, at the near-to-the-metal level where they are usually coped with. When working with terms, their type is represented by the concrete class their instances belong to; and that type can be inspected using the appropriate predicate from the public interface of the Term class, depicted in Fig. 4: alongside methods like isVar, isStruct, isNumber, there also are predicates for checking more specific types like lists, atoms, and compound terms. Then, whenever a term represents a Prolog variable and that variable is bound, the actual term linked to the variable can be obtained by means of the getTerm method.

Classic, higher level criterions for comparison between Prolog virtual machines still applies, though: for instance, the division into the categories of structure sharing [12] or structure copying [13] systems, based on the distinction on how compound terms are represented. Instead of following the molecule scheme, composed by a reference to an array containing the values of the term’s variables,
and a reference to the term’s non-variable representation, fashionable in early systems but still in use even in object-oriented interpreters [9], tuProlog follows the structure copying representation, distinctive of most modern systems including WAM based ones, where all compound terms are made up of an element identifying the main functor and an array containing the term’s arguments. The unification process is typically carried out faster in Prolog machines using the structure copying representation.

5 Unification

The relevant steps in the unification algorithm for the Prolog programming language can be described as follows:

- two atomic terms (e.g. numbers, strings, constants) unify if they are equal;
- two unbound variables unify, and they are bound together;
- an unbound variable and an atomic or compound term unify: that term is bound to the variable, and the variable references it;
- two compound terms unify if their functors are equal both in name and in arity, and all their arguments unify.

The definition of unification and the data representation are mutually influential: this is so much true also for object-oriented interpreters, where the unification algorithm is typically distributed amongst the various classes representing the elements of the Prolog language. Each term knows how to unify itself to another term, and how to provide significant information to the engine during the process.

The basic mechanism used by tuProlog to perform unification between two terms is to annotate all the unified variables in a list. That list is then added to the trailing variables list, stored in the core engine, which keeps track of the substitutions carried out during each step in the demonstration process. When it is necessary to perform deunification, the list containing the unified variables is traversed, and each variable gets freed.

The details of each term object’s behavior are spread among the abstractions which the code implementing the unification algorithm is distributed over. The first method to be always called is however the unify method belonging to the Term class, taking as parameters the Prolog instance which the unification is performed within, and the term which the unification is performed upon. That method will subsequently call another unify method on the first of the two terms. It receives as arguments two (initially empty) lists which will respectively contain unified variables belonging to the first term and those belonging to the second term, and the second term which the first term has to be unified with. The method performs a check on the received term: for example, if a unification

3 Below the API, at the implementation level, lists in tuProlog are indeed an exceptional case: a field for the ' .' functor is used, but instead of being memorized using a sequential structure, they are implemented by the Lisp-like head and tail recursive scheme.
between two Struct instances is requested, the equality of functor and arity between the terms is verified, then unification is propagated on the compound term’s arguments; if a Struct is to be unified with a variable, then the unify method of the variable object is called, passing the Struct instance as an argument and inverting the order of the two lists which will contain unified variables from the two terms. A similar check on the type of the term to be unified is performed also in the unify method of the Var class. If that term is a Struct, the occur check is performed. Whenever that check is not successful, or the unification fails for another reason, it is not necessary to deunify the variables already annotated in the lists, because they will be taken care of by the backtracking phase, always consequential to a failed unification. Alongside the more complex algorithms for the unification of Struct and Var objects, the straightforward behavior of unification for numbers and atomic Prolog terms takes place.

Despite being discarded in the early times for efficiency reasons [1], the occur check is introduced in tuProlog and always executed because, instead of raw performance, it is considered more interesting to avoid the rise of problems due to the presence of the same variable in a structure as the variable which the structure is going to be bound to. However, it must be noted that optimizations are still investigated when they reside at the appropriate abstraction level and do not interfere with the correctness of the demonstration process. In fact, to avoid the costly comparison of strings as names of terms at every unification step, a map of symbols has been introduced. Whenever a term in the logic theory or in the user’s query is recognized, the string corresponding to that term’s functor is inserted in the map; if the map already contains the same string, then the string corresponding to the term’s functor gets substituted by the one in the map. In that way, during unification it is not necessary to compare string objects by means of the equals method, but it simply suffices to compare the references of the two objects by using the == Java operator.

6 The Resolution Tree

As a further implementation detail, it is interesting to have a glance at how the resolution algorithm, the real workhorse of any Prolog system, is run from within tuProlog’s inferential core, and which data structures are used to perform execution, support backtrack, and store trailing variables during the process.

When the Finite State Machine described in Sect. 3 starts, the first data structure to be built is the execution context for the query whose solution has been asked to the engine, represented by the ExecutionContext Java class. One of its instances holds a reference to the subgoal executing in that context, one to the clause which the context has been constructed from, and one to the body of that clause, represented under the form of a tree of predicates rather than a nested recursive structure, for efficiency reasons. From this tree, subsequent terms are extracted and evaluated, in different modes depending on their nature: if a term represents a primitive predicate or evaluable functor, it is directly executed in the Goal Evaluation state without the need for building a new
ExecutionContext instance; instead, if a term represents a Prolog predicate to be matched with the head of a clause or set of clauses from the logic theory contained in the engine, a new ExecutionContext is created in the Rule Selection state, referencing the first matching clause found. This new execution context holds a reference to the previous context as its parent context; by means of such linking, during the demonstration process a tree structure gets created, known in the literature with the name of proof tree.

The root of a proof tree corresponds to the initial query; its children are the subgoals composing that first goal. In every subtree of the proof tree, a node represents the head of a clause, and its children are the subgoals composing the body of that clause. Leaves in the tree are facts: only if the leaves for a given subtree are true, the demonstration for that subtree can successfully end. Otherwise, a failure is encountered, and the process must be backtracked to the latest multiple choice point: the state of the tree must be restored as before the first call to the failing predicate, then another possible alternative for that predicate is chosen, and a different structure for that subtree is created while the demonstration process continues. The Finite State Machine core of tuProlog always works on the latest active node, referenced by the currentContext field in the Engine class. When the demonstration fails on a certain node, the core must backtrack to a node previously executed, from where an alternative path can be followed. In the Backtrack state, the engine substitutes the chain of contexts it is currently working on with a previously used chain identified by the selected choice point. Since references to choice points are kept in a ChoicePointStore within the Engine, the execution can directly jump to the latest choice point and prune the failing path instead of visiting each node backwards. Following the substitution, variables which were unified during the execution of nodes from the selected choice point to the failing node must be freed, to restore a coherent state in the engine. Those variables are tracked in a special environment created for choice points by the ChoicePointContext class.

Data structures storing choice points and trailing variables are shaped with stack semantics: ChoicePointContext instances hold a reference to the previous choice point encountered during execution; trailing variables are stored as lists contained in a list carried by each ExecutionContext. On the other hand, since the substitution of the chain of contexts is only made when a backtracking occurs, it must be noted that the proof tree is not internally reduced to a stack containing the current trail only, as instead happens in other Prolog systems, such as the Warren Abstract Machine. In fact, alongside the currently successful path in the tree, which is always stored in memory, also paths containing open choice points are maintained in memory as well, since one of them could be promoted to be the current execution path whenever a backtracking occurs. References to parts of the proof tree are discarded, and memory is cleared with help from the Java Virtual Machine’s underlying garbage collection mechanism, only when a failing node in encountered, and an alternative path must be tried, starting from the latest choice point. The peculiar way this proof tree structure evolves,

---

4 Facts or primitives, in the special case of the tuProlog engine.
with its nodes linked from child to parent only, could benefit the traceability of
the demonstration process for debugging purposes: by storing references to each
leaf, no branch would be lost to garbage collection, and the whole tree could be
inspected even during runtime.

7 Conclusions and Future Work

tuProlog’s architecture, based on a minimal core implemented as a Finite State
Machine, which a set of managers operates around, by handling control of sensi-
bile parts of the engine, has shown how sound design principles can lead to flexible
and modular systems. In particular, engineering properties like configurability
and extensibility have been carried over from the whole engine to its core, and
a clear path for deep traceability has been laid. Future work will be devoted to
the exploitation of those properties, by focussing on the independent evolution
of the different subsystems which the tuProlog engine has been modeled and
divided into.

References

SIGPLAN conference on History of programming languages, New York, NY, USA,
Tool. DAI Research Reports 39, 40, University of Edinburgh (1977)
International (1983)
mierung. (1994) 1–15
6. Tarau, P.: A Simplified Abstract Machine for the Execution of Binary Meta-
programs. In: Proceedings of the Logic Programming Conference ’91, ICOT, Tokyo
(1991) 119–128
applications and infrastructures. In Ramakrishnan, I., ed.: Practical Aspects of
8. Denti, E., Omicini, A., Ricci, A.: Multi-paradigm Java-Prolog integration in
9. Lanovaz, D., Szafron, D.: An Object-Oriented Inference Engine for Prolog. The
(1994)
In Melzer, B., Michie, D., eds.: Machine Intelligence. Volume 7. Edinburgh Univer-
sity Press (1972) 101–116