The Architecture and Design of a Malleable Object-Oriented Prolog Engine

Giulio Piancastelli
DEIS, Università di Bologna
via Venezia 52
47037 Cesena (FC), Italy
giulio.piancastelli@unibo.it

Alex Benini
DEIS, Università di Bologna
via Venezia 52
47037 Cesena (FC), Italy
alex.benini@unibo.it

Andrea Omicini
DEIS, Università di Bologna
via Venezia 52
47037 Cesena (FC), Italy
andrea.omicini@unibo.it

Alessandro Ricci
DEIS, Università di Bologna
via Venezia 52
47037 Cesena (FC), Italy
a ricci@unibo.it

ABSTRACT

The implementation of Prolog systems has a long history, from the first interpreter written in 1972 to de facto standard model of the Warren Abstract Machine. Although many architectural variations have been proposed, object-oriented design was left mostly unexplored, favoring other factors such as execution time and memory storage optimizations. However, today complex software systems are typically built as aggregates of heterogeneous components, where logic programming may effectively help facing key issues such as intelligence of components and management of interaction. In this scenario, implementation of logic languages could just aim at reasonable – rather than maximum – efficiency, requiring instead configurable and flexible architectures to allow for extensions and tailoring for different application domains. tuProlog is an object-oriented Prolog engine which has been designed to feature a malleable architecture at its core, and to exhibit the typical properties of basic components for complex dynamic systems and intelligent infrastructures—such as easy deployability, lightness, and configurability. In this paper, we first describe tuProlog’s malleable architecture, composed by a set of managers controlling sensible parts of the system, and operating around a minimal interpreter shaped as a Finite State Machine. Then, we support the malleability claim by discussing two possible architectural extension of the engine.

Categories and Subject Descriptors
D.3.4 [Programming Languages]: Processors—Interpreters; D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures; D.1.5 [Programming Techniques]: Object-oriented Programming; D.1.6 [Programming Languages]: Compilers—Interpreters

General Terms
Design, Experimentation, Languages

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

1. INTRODUCTION

The implementation of Prolog systems has a long history, 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 a year later [2, 12]. The reasonable performance of that system helped convince 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 [14], the first Prolog compiler, with the aim of optimizing performances by simplifying basic operations such as unification and backtracking. By 1983 Warren had also developed an abstract machine, composed of a high-level instruction set and a memory model, which has become the de facto standard implementation technique, named the Warren Abstract Machine (WAM) [15].

Along the lines of the WAM, many different variations have been proposed, from the Vienna Abstract Machine models [9] to logic engines based on the continuation passing style model of Prolog [11]. However, despite the advent of object-oriented programming, the architectural debate did not further develop to include object-oriented design principles, because the field of Prolog engine construction evolved by favoring factors such as execution time and memory storage optimizations. Indeed, we are aware only of the work by Lanovaz and Szafron, who described an object-oriented Prolog inference engine written in Smalltalk for Gödel, the Graphically Oriented Development Environment for Logic programming [10].

The creation of complex systems in network-centric and other modern computing models is drifting from monolithic architectures to aggregates of components where a heteroge-
neous set of programming paradigms is typically employed. Such systems need to successfully face the fundamental challenges of intelligence and interaction. Logic programming languages have always enabled the development of intelligent components, and they also proved to be effective both as communication and coordination languages [3]. This suggests that logic languages may play a key role in building complex systems, such as dynamic Internet infrastructures, for application domains where intelligence and interaction are regarded as the core issues. The scenario we envision has thus two important consequences on implementations of logic programming languages: they could aim at achieving just reasonable, rather than maximum, efficiency; they would need configurable and flexible architectures, to take into account extensions and tailoring applied in order to satisfy different application domain requirements.

\t\text{tuProlog}^1 \text{ is an object-oriented Prolog engine which has been designed to build intelligent components for dynamic Internet infrastructures [4]. The purpose of our research was to develop a malleable architecture for tuProlog’s inference core, as the result of applying sound engineering practices such as object-oriented design, reuse of established community knowledge in the form of patterns, loose coupling of composing elements, modularity, and a clear and clean separation of concerns. Besides, we also needed tuProlog to exhibit those engineering properties, such as easy deployability, light-weight, and configurability, which are typical for basic bricks of complex dynamic systems and intelligent infrastructures.}

\t\text{In this paper, we present the tuProlog engine’s malleable architecture, composed by a set of managers controlling sensible parts of the system and operating around a minimal core shaped as a Finite State Machine. Then, to support the malleability claim, we discuss the design of two possible architectural extensions of tuProlog.}

\section{2. FEATURES OF \text{tuProlog}}

\text{tuProlog is a Java-based, light-weight, easily deployable, and configurable Prolog engine, written in an object-oriented fashion and featuring a malleable architecture at its inference core. Malleability}^2 \text{ is an architectural property that refers to the ease with which a change can be made to a software architecture [7]. Malleability is not an atomic property, but can be further decomposed into evolvability, extensibility, and customizability. Evolvability represents the degree to which an architectural component can be changed without negatively impacting other components, and it depends on how well the architectural abstractions are enforced by the design. Extensibility is defined as the ability to add functionality to a system, and is favored by loosing the coupling between architectural components. Customizability represents the ability to specialize the behavior of an architectural component, in such a way that a customized component can be used by a client of its specialized services without adversely impacting other clients of that component. Customization may also lead to the construction of simple and scalable architectural components, by adequately separating the concerns of basic services from specialized functionality.}

\text{In the first place, tuProlog’s architectural malleability derives from the decomposition of relevant functionalities into simple manager components with a single and well determined responsibility. By separating the concerns and loosing the coupling between parts of the engine, both extensibility and customizability are favored. Secondly, the realization of the inferential core as a Finite State Machine helps both evolvability, by enforcing that each state abstraction represents a well known step in the resolution algorithm, and extensibility, by allowing the machine to be augmented with new states to tailor the interpreter’s inner workings. Finally, evolvability and customizability are also privileged by modeling as first-class objects fundamental entities of a Prolog virtual machine, such as the execution context of a node in the resolution tree.}

\text{Alongside architectural malleability, software components also need to exhibit engineering properties such as easy deployability, lightness, and configurability, when they are intended to be used as infrastructure elements. tuProlog owes its deployability to Java, since requirements for installation simply amount to the presence of a standard Java Virtual Machine. tuProlog also favors lightness by being a minimal Prolog engine, available in a single JAR file. Finally, tuProlog features configurability by means of libraries, a simple yet powerful mechanism to load and unload predicates, functors, and operators, both statically and dynamically.}

\section{3. ARCHITECTURE OF \text{tuProlog}}

\text{Two main requirements dictated the initial sketch of the architecture of tuProlog: a minimal Prolog virtual machine, available as a self-contained object, featuring a simple interface; and the need for configurability 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 Fa¸cade [8] pattern. The division into managers, for the inferential core, theories, libraries and primitive predicates, has been carried out to allow subsystems to grow as independently as possible from each other, and to separate the concerns about tuProlog’s configurability, supported 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 Fa¸cade design pattern is employed also with the aim of reducing the number of objects that clients have to deal with, and of making the system easier to use.}

\subsection{3.1 Libraries Management}

\text{The Library Manager architectural component 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 resolution 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}

\footnotesize{\textsuperscript{1}Available at http://tuprolog.alice.unibo.it
\textsuperscript{2}In the literature, malleability is sometimes also referred with the term modifiability.}
basic to be defined elsewhere, such as \texttt{fail/0}; and the ones which need to be defined near the core for efficiency reasons, such as \texttt{\textasciitilde, \textasciitilde/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 contained in 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 systems, 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 only those predicates which can never be unloaded from the engine.

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: \texttt{(i) BasicLibrary}, which defines some basic predicates and functions usually found in Prolog systems, with the exception of I/O predicates; \texttt{(ii) IOLibrary}, which provides some of the standard Prolog I/O predicates, such as \texttt{write/1, read/1} and \texttt{nl/0}, further separated by other functionalities, so as to enforce orthogonality between interaction and computation; \texttt{(iii) ISOLibrary}, which defines standard ISO predicates not defined in the previous libraries and not concerning the I/O realm; and \texttt{(iv) 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 muddle of the two paradigms \cite{5}. 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 library under the form of a Java string, to be added to the engine’s configuration whenever the library is loaded by the manager subsystem. In tuProlog parlance, Prolog library predicates are known as \texttt{library predicates}, while Java library predicates are called \texttt{primitive predicates}, or simply \texttt{primitives}. Prolog predicates coming from a program fed by a user to the engine are instead known as \texttt{user-defined predicates}.

### 3.2 Primitives Management

When a Prolog theory is read by the engine, the Primitive Manager component identifies the predicates and functors contained in the theory 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. Libraries and user-defined predicates are taken care of by the inferential core and its Prolog execution algorithm; primitive predicates 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.

Each library is able to provide a reference to its methods defining primitive predicates and evaluable functors via Java reflection mechanisms. When primitive predicates and evaluable functors have to be executed, the engine takes care of running the primitive by invoking the Java method corresponding to it, exploiting again the Java reflection facility.

### 3.3 Theories Management

The tuProlog engine consults a Prolog program by running the program’s text through the parser, which recognizes the syntax of correct sentences in the Prolog programming language and creates objects corresponding to the identified terms. Interleaved with the parser, the Theory Management subsystem 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, which holds a reference to the term object representing a Prolog clause, and supplies additional data, such as a flag indicating if that clause is dynamic (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). Instances of that helper class are stored in lists of clauses belonging to the same family, \textit{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 resolution process; the hashing ensures fast access to members of the same family.

For optimization purposes, the body of each clause is immediately decomposed in subgoals as soon as the clause itself is recognized: by early decomposition at construction 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 meta-variable facility of 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.

### 3.4 Engine Management

That part of the public interface of the Prolog engine that contains operations dealing directly with the invocation of routines in the inferential core is administered by the Engine Manager subsystem. The aim of this architectural component is to loose the coupling between the façade of the system and the machine realizing the core engine.

Apart from defining some functionalities for the Prolog Finite State Machine implementing the façade of the system, the engine management 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

\footnote{For further details about libraries, including a list of defined predicates and guidelines on developing new libraries, the interested reader should refer to the Guide included in tuProlog’s distribution.}
contain a Prolog theory, and that theory could include a initialization/1 directive, to be executed as soon as the Prolog text in the library has been 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 state 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.

4. THE FINITE STATE MACHINE CORE

The inferential core of tuProlog is built in the form of a Finite State Machine (FSM), whose basic architecture is depicted in Fig. 1. The machine is composed by (i) an initial state, assumed as the solving procedure starts, (ii) four main states representing the activities performed during the resolution process, and (iii) four final states which identify the different ways a demonstration may be ended.

4.1 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 by setting up an object which represents the execution context for the current subgoal. That subgoal is chosen by the next state, to which the Init state immediately passes control.

The Goal Selection state fetches the next goal to be executed from the subgoal list. If such a goal does not exist because the resolvent is empty and the demonstration process has 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 meta-programming behavior. The FSM then changes its state to take care of the next step in its execution algorithm: goal evaluation.

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 the main functor of the extracted subgoal does not represent a primitive, its evaluation can only be performed by browsing the logic theory contained in the engine and selecting a compatible clause. These actions are performed in the Rule Selection state. Whenever an error condition is reached during the evaluation of a goal, the machine stops the computation in the HALT state.

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 Goal Selection.

Once entered the 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 FALSE state to immediately make the demonstration fail if that set is empty. Variables and the resolvent are then returned back to their pristine state. 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 Goal Evaluation, to let the resolution process restart.

4.2 Design

The actual design of the finite state machine is realized using the State pattern [8]. This design 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 tuProlog system’s case, it is the inferential core Engine, started by the Engine Manager component, 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 State class, providing the abstract doJob method which contains the state’s behavior in the concrete subclasses. From State all other classes representing the core’s states are derived: StateInit, StateRuleSelection, StateBacktrack and so on, with a clean name pattern matching state names in the FSM.4 At a finer detail, the pattern is implemented by having the Engine class re-

4Since end states do not feature substantial behavior, they
membering the next state it must shift to, and having state classes directly manage the transitions between each other.

5. DISCUSSION AND EXPERIMENTS

The architecture of tuProlog we have described in Sect. 3 and 4 leads to some advantages from the point of view of malleability and related non-functional properties, such as evolvability, extensibility, and customizability. We have obtained a clear separation of concerns in tuProlog’s internals, by means of both loose decomposition into managers and the inferential core’s modeling as a Finite State Machine. This opens various possibilities: on the one side, for both extension and customization of single managers; on the other side, for both evolution of the Finite State Machine with concepts that fit the current model, and customization of the machine’s behavior at the single state grain.

Amongst potential modifications to the engine’s architectural components, some significant examples come to mind: evolution of the Theory Manager subsystem for the inclusion of modules or labeled theories; introduction of the handling of Prolog errors into the core’s execution flow; extension of appropriate parts of the system to support constraint programming, defeasible logic, or argumentation. Besides, directly considering the Finite State Machine architectural properties, more opportunities for exploiting malleability are offered, such as customization of the Backtrack, Rule Selection, and Goal Selection states to allow 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.

To provide evidence for the claim of architectural malleability, in the following sections we discuss the design of two from the potential modifications enlisted above: the introduction of Prolog error-handling mechanisms into the engine, and the extension of tuProlog towards argumentation.

5.1 Adding Exceptions

As defined by the ISO Standard, a Prolog error is a special circumstance which causes the normal process of execution to be interrupted [6]. An error-handling mechanism should be able to detect errors and, instead of stopping the program execution to a handler receiving a value that describes the error. The Prolog error-handling mechanism follows two basic principles in exceptions modeling: the error confinement principle, by means of the catch/3 predicate, and the atomic jump principle, provided by the throw/1 predicate [13]. First, the error-handling mechanism should be able to confine the error so that it does not propagate to the whole program. Assuming that the program is written in a modular programming language (i.e. made up of interacting components organized in a hierarchical fashion), the error confinement principle states that an error in a component should be catchable at the component boundary; outside the component, the error is invisible or reported in a nice way. The error-handling mechanism causes a jump from inside a

![Figure 2: The Finite State Machine extension to support Prolog error handling.](Image 359x627 to 514x738)

component to its boundary. This jump should be atomic: the mechanism should be able, in a single operation, to exit from arbitrarily many levels of nested execution contexts, previously created by predicate calls and sequential compositions.

To validate the malleability claim, we propose to introduce a simple extension to tuProlog’s architecture that follows the Prolog error-handling model. The insertion of Prolog error mechanisms in the execution flow of tuProlog could be modeled as an evolution of the engine’s core by introduction of another state in the Finite State Machine, as illustrated in Fig. 2. This new Exception state would be assumed by the machine from the Goal Evaluation state, whenever an error occurs during the execution of a Prolog subgoal and that subgoal gets substituted by a throw/1 invocation. The job of the Exception state should be to carry out the procedural side effect of throw/1: causing the normal flow of control to be transferred back to an existing call of catch/3 which matches the thrown error. Keeping the Prolog term representing the error, the Exception state would need to trigger a backwards visit to the resolution tree composed of execution context objects, in order to find the appropriate catch/3 subgoal, which needs to feature a second argument unifiable with the error to be caught. During the search, each traversed execution context would be pruned, so as not to be executed or selected by backtracking anymore.

After identifying the execution context with the correct catch/3 subgoal, the Exception state would need to eliminate that context from the resolution tree, and substitute it with a new node containing the error handler as defined in the third argument of that catch/3 subgoal. The Exception state would also need to prepare the handler for execution, by preserving the substitutions applied during unification of the error thrown by throw/1 and the term argument of catch/3 to be unified with the error. Finally, the machine would then switch to its Goal Selection state, or halt the execution by moving into the Halt state, in case the error is not to be managed.

Apart from implementing catch/3 and throw/1 as built-in predicates, due to the need of keeping them near the engine’s internals, no other change is required to add error-handling capabilities to tuProlog. This case study has therefore verified the malleability of tuProlog’s architecture, by way of its core’s evolvability: modifications have been kept within the Finite State Machine boundaries, and the engine’s architectural abstractions have been properly designed to endorse new programming concepts that were not initially devised.
5.2 Extending towards Argumentation

Argumentation is a resolution process based on the exchange and evaluation of interacting arguments to support operations of non-monotonic reasoning [1]. This process generates arguments and counter arguments over an inconsistent knowledge base, determine the acceptability of those arguments, and construct proofs conforming to one of a wide range of semantics.

In the context of the ASPIC research project, a non-monotonic reasoning component supporting argumentation has been prototyped by re-engineering tuProlog and implementing an algorithm based on an argument game approach to construct proofs of acceptance [1]. This algorithm interprets argument games between two players – a proponent and an opponent, sharing the same knowledge base – as constructing proofs of acceptance using a dialectical structure. The proponent starts with a claim to be proved: she attempts to build an admissible set of arguments to support the claim, and endeavors to defend any argument against attacks coming from the opponent. The proponent wins the game if all the attacking arguments have been defeated; otherwise, she loses if the opponent is able to find an attacking argument that cannot be defeated.

To support the assertion of tuProlog’s architectural malleability, we propose a design of the argumentation engine that better fits within tuProlog’s architectural abstractions than the design sketched in the ASPIC prototype. Three main modifications are needed: (i) support for recognizing degree of belief values, defined alongside logic clauses that express propositions in the knowledge base; (ii) a hierarchy of classes representing trees of arguments to support or attack a claim; and (iii) mechanisms to execute argumentation queries, establishing whether a claim can be supported within the knowledge base, and analyze results, determining whether the arguments are acceptable.

The degree of belief needs to be both recognized and exploited by the engine. To this end, simple customizations can be performed on the components involved in the process of manging clauses and managing the knowledge base: the parser, the theory management subsystem, and the data structure where each clause is stored would all need to take into account the existence of the degree of belief attribute in every element of argumentation logic programs. Hierarchy of arguments may be represented by extending tuProlog with a taxonomy of classes structured around the Composite design pattern [8], 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.

The primary goal of the argumentation process with dialectic semantics is to prove a claim by building an admissible set of arguments, in a dialogue between an attacking opponent and a defending proponent. Two main responsibilities emerge from this requirements: first, the construction of the argument tree, occurring twice during a demonstration, once when supporting the original goal and once when attacking all supporting arguments; second, the execution of multiple solving processes that need to act upon both the proponent and opponent proofs on the same logic theory already existing into the engine. To account for the activity of argument tree construction, tuProlog could be extended with a new Argument Manager subsystem: it would build the argument tree – with possibly multiple solutions attached – and contain data structures to store a list of all terms that have been attacked and the strength they were defeated by. This new management component would need to ask for supporting and attacking executions of tuProlog’s Finite State Machine core, then to pick results and store them internally, in order to know by comparison the final query result. Within tuProlog’s architecture, responsibility for nesting multiple solving processes is already assigned to the Engine Manager component, as described in Sect. 3.4. However, this subsystem would need to be customized, in order to account for specific interactions with the Argument Manager component.

The design of an argumentation engine based on an existing Prolog interpreter has a profound impact on its foundation. Nonetheless, this case study supports tuProlog’s architectural malleability in many ways. The well-factorized layout of tuProlog’s architecture has favored simplicity in crafting the design of the argumentation engine, despite the many modifications needed. Also, loose coupling of architectural elements allowed changes to be kept within components’ boundaries. Finally, customizability and extensibility of tuProlog’s subsystems proved themselves essential in showing how tuProlog can be adapted to support evolutions of the logic programming paradigm.

5.3 A Note on Performance

One of the perils of introducing many layers of architectural abstractions in a system is the risk of penalizing overall performance. Need or usefulness at the very least should drive decisions in inserting new abstraction layers in a system, especially when a layer gets traversed every time a critical operation is performed. Performance of a system should also be adequate to the envisioned context where the system will be employed.

tuProlog has been designed to enable the development of intelligent components for complex systems, such as dynamic Internet infrastructures, where coordination and interaction are regarded as core issues. This scenario enforces the need for architectural malleability, and we have shown the usefulness of such a property in the proposed evolutions of tuProlog’s design, as outlined in Sect. 5.1 and 5.2. tuProlog’s application context would also call for reasonable, rather than maximum, efficiency: in Table 1, we have measured tuProlog’s performance against a set of well-known Prolog benchmarks, and confronted the results with other Java Open Source Prolog interpreters.

Table 1 lists the results of running classic Prolog benchmarks (including naive list reversing, prime numbers generation, quicksort, and the queens problem) against tuProlog. Comparisons are made with results from: the most important Prolog engines implemented on top of the Java Virtual Machine, that is JIProlog, JLog, and JavaLog. Benchmark tests shows that JavaLog is the worst performer, while JLog outperforms other implementations in every benchmark. In the middle between these two extremes, tuProlog competes with JIProlog for the second place, beating it on six of the nine benchmark tests.

---

3Developed by Analía Amardi and Alejandro Zunino at the Universidad Nacional del Centro de la Provincia de Buenos Aires, and downloadable from http://www.exa.unicen.edu.ar/isistan/javalog.html.
As far as performance is concerned, we can therefore assert that tuProlog’s goal for reasonable efficiency has been accomplished. The introduction of architectural abstractions, such as managers and the Finite State Machine in the engine’s core, does not dramatically impact on overall performance, and the benefits of malleability in tuProlog’s architecture exceed the efficiency loss with respect to other Java-based engines.

6. CONCLUSIONS

tuProlog’s malleable architecture, based on a set of managers operating around a minimal core implemented as a Finite State Machine, has shown how the application of sound design principles can lead to the construction flexible and configurable systems. In particular, we showed how engineering properties such as customizability and extensibility, carried over from the whole engine to its core, allow for evolutions of many sorts to be developed upon tuProlog’s well-laid foundations.

From the scientific viewpoint, it has been our purpose to try and stimulate the architectural debate further in the object-oriented direction, since we believe that the basic bricks of complex dynamic systems and intelligent infrastructures, for the present and the foreseeable future, will be built upon object-oriented platforms such as the Java Virtual Machine or the Common Language Runtime.

7. REFERENCES


