An IPC-based Prolog design pattern for integrating backward chaining inference into applications or embedded systems

Li Guoqi, Shao Yuanxun, Hong Sheng *, Liu Bin

Science and Technology on Reliability and Environment Engineering Laboratory, School of Reliability and Systems Engineering, Beihang University, Beijing 100191, China

Received 14 December 2013; revised 23 May 2014; accepted 6 August 2014
Available online 22 October 2014

Abstract Prolog is one of the most important candidates to build expert systems and AI-related programs and has potential applications in embedded systems. However, Prolog is not suitable to develop many kinds of components, such as data acquisition and task scheduling, which are also crucial. To make the best use of the advantages and bypass the disadvantages, it is attractive to integrate Prolog with programs developed by other languages. In this paper, an IPC-based method is used to integrate backward chaining inference implemented by Prolog into applications or embedded systems. A Prolog design pattern is derived from the method for reuse, whose principle and definition are provided in detail. Additionally, the design pattern is applied to a target system, which is free software, to verify its feasibility. The detailed implementation of the application is given to clarify the design pattern. The design pattern can be further applied to wide range applications and embedded systems and the method described in this paper can also be adopted for other logic programming languages.

© 2014 Production and hosting by Elsevier Ltd. on behalf of CSAA & BUAA.

1. Introduction
Prolog is a general purpose logic programming language and is the most popular one among such kind of languages. Although it is not trendy as it used to be, Prolog is still one of the most important candidates to build expert systems and AI-related programs. Recent hot topics on Prolog are its integration with ontology, prolog specification of giant number arithmetic, portability in different implementations of Prolog and conjunction with programs developed by other languages.

Compared with imperative programming languages, such as C/C++ and Java, Prolog is declarative and driven by rules matching. Prolog concentrates on how to describe problems rather than how to solve them step by step. From the sense of contemporary computing, Prolog has the following two outstanding advantages.

(1) Higher development efficiency in solving AI-related programs. There is small semantic gap between AI algorithms and Prolog. Here, semantic gap refers to the
difference between the natural representation of some knowledge and the programmatic representation of that knowledge. Additionally, as generally recognized, Prolog is simple and easy to grasp.

(2) Lower costs in validation and verification. For programs with the same functions, the code is usually much shorter written in Prolog than in imperative languages. With the increasingly high demand for software quality, it is a great advantage in many ways, especially for critical systems.

Certainly, lower operational efficiency is its shortcoming. However, it seems to be insignificant, because hardware resources, such as CPU computing power and memory capacity, are sufficient enough nowadays.

Furthermore, knowledge-based techniques are currently reported to be applied in real-time control systems, which are typical and popular embedded systems. Consequently, Prolog has potential applications in autonomous cars, home medical devices and unmanned aerial vehicles (UAV), etc., which are hot topics currently and are hopeful to enter our daily life in recent future.

Despite of these facts, Prolog is not suitable to develop many kinds of components, such as data acquisition and task scheduling, which are also crucial. Accordingly, it is attractive to combine Prolog and applications or embedded systems together. However, coincidently and unfortunately, it is a bottleneck. To break this bottleneck, this paper brings forward an IPC (inter process communication)-based Prolog design pattern for integrating backward chaining inference into applications or embedded systems. Backward chaining, or backward reasoning, is an inference method that can be described as working backward from the goals. It starts with a list of goals (or a hypothesis) and works backwards from the consequent to the antecedent to see if there is data available that will support the antecedent to see if there is data available that will support the antecedent to see if there is data available that will support the antecedent to see if there is data available that will support the antecedent to see if there is data available that will support the antecedent to see if there is data available that will support the antecedent to see if there is data available that will support the antecedent to see if there is data available that will support.

In essence, our method obtains the result of Prolog by IPC, instead of by capturing the output of Prolog console. The method, whose schematic diagram is shown in Fig. 2. Compare with the former figure, the gray rectangles are different parts. For illustration. Finally, we draw conclusions and address our future works.

2. Principle and definition of the design pattern

2.1. Principle

SWI-Prolog provides flexible and fast interface to C/C++ and Java, which allows for calling both ways. However, it is not enough. As a full featured programming language, Prolog is designed as a standalone system. Concrete Prolog implementations, such as SWI-Prolog, GUN Prolog and Amzi! Prolog, usually provide independent consoles, multithread mechanisms, build-in predicates, etc. When calling a Prolog program from C/C++ or Java through the interface of SWI-Prolog, additional threads or processes will be created and the results of the Prolog program will be output to the console of SWI-Prolog, instead of returning to C/C++ or Java. Light weight systems, such as Drools and Clips, are more suitable than Prolog for embedding, because their output is easier to be obtained or easier to be parsed than Prolog. However, Prolog has more powerful build-in predicates and large amount of the existing codes, which make it valuable to be integrated.

Consequently, in 2007, Peretyatkin provided a method to integrate Qt with Prolog. The schematic diagram of the method is shown in Fig. 1. The “Output to Prolog console” is captured and parsed so that the result of the Prolog program could be extracted and transferred to C/C++. However, the method has two problems.

(1) Case specific. For different Prolog programs, Qt part must provide different parsers for each of their input and output.
(2) It is error-prone to capture the output to the Prolog console and it is difficult to verify the parsers. There is no mechanism to guarantee that the output of the Prolog program is predictable, namely, it is difficult to add constraints to the output. Peretyatkin’s method was intended for education, so it is fine. However, if for industrial usage, safety must be taken into account.

To solve the problems, this paper presents an IPC-based method, whose schematic diagram is shown in Fig. 2. Compare with the former figure, the gray rectangles are different parts. In essence, our method obtains the result of Prolog by IPC, instead of by capturing the output of Prolog console.

In fact, IPC is not a new topic for Prolog. As early as in 1993, the method of communication between prolog and an external process was put forward as a patent and in 1996, process to process communication in prolog was described in detail in a journal paper. Practically, in SWI-Prolog, IPC is supported in the form of Prolog library. However, in former works, IPC was used to improve the efficiency of Prolog engine or for distributed computing and it has not been reported that IPC is applied to integrating Prolog with applications or embedded systems. In this paper, IPC-based method is introduced to integrate backward chaining inference into applications or embedded systems and a design pattern is derived for reuse.

Actually, IPC library is an extension of Prolog. As shown in Fig. 2, the “IPC extension” is a set of predicates. When inferring to the predicates, they will trigger IPC with C/C++ and transfer the result of the Prolog program to C/C++. The
module (facts, [parent/2]).

The second line of the facts.pl defines this module named "facts". Module and its source file share the same name. The "[parent/2]" in the line indicates that "parent" is a binary predicate and could be imported to other modules. Lines following the second define many facts with the predicate.

The algorithms.pl imports the "facts" module mentioned above with the third line. It imports all the predicates listed in the bracket of the second line of facts.pl. Then, the predicates are all visible and could be reused in the algorithms.pl. The fourth line of the algorithms.pl imports another kind of module, named "foreign predicates", which is listed in the following. Unlike the "facts" module, it is a foreign module. A module using foreign libraries is called a foreign module. Look at the third line of the foreign predicates.pl. It imports a foreign library, named "yoyo_reasoner". As mentioned above, it is an IPC extension of Prolog. Foreign library means that the predicates in the library are implemented by other languages, such as C or C++.

You could just ignore the lines following forth of the algorithms.pl, which will be described in the next section.

Fig. 2 Schematic diagram of IPC-based method to integrate Prolog with applications.

"IPC extension" is implemented based on the mechanism of Prolog calling C/C++. For the sake of concise, in the following descriptions, "yoyo_reasoner", a simple IPC extension, implemented by our team, is used, instead of the complicated IPC library of SWI-Prolog.

Compare with the former integrating method illustrated in Fig. 1, the IPC extension is case independent and could be reused. The communication between Prolog and C/C++ depends on the "IPC extension". The constraints on the predicates of the "IPC extension" guarantee that the data transferred from Prolog to C/C++ are also be constrained, so it is safe for industrial use. This point will be clarified in detail in the following sections.

2.2. Definition

Before defining the Prolog design pattern, many related Prolog definitions should be clarified.

Prolog module encapsulates a set of predicates and defines an interface. Modules can import other modules, which makes the dependencies explicit. Actually, module is a mechanism to modularize Prolog programs and control the visibility of the defined predicates in the module. For example, look at the following Prolog source code file facts.pl:
We implemented a foreign library in C++ and compiled it as a dynamic link library (DLL) named "yoyo_reasoner.dll". Actually, the third line of the foreign_predicates.pl loads the DLL and imports the predicates in the library. There are two predicates in the foreign library: "add_to_buffer/1" and "tcp_send/1". These two predicates are different from conventional ones. When inferring to these predicates, they trigger C++ programs in yoyo_reasoner.dll. Corresponding to "add_to_buffer/1" and "tcp_send/1", there are two C++ functions, "pl_add_to_buffer" and "pl_tcp_send". The former one adds the parameter of the predicate "add_to_buffer/1" to a buffer and the latter creates a socket, connects it to the server and sends the data in the buffer to the server. Here, we used TCP-based, namely socket-based, IPC. D-Bus and other kinds of IPC methods are also feasible.

The "yoyo_reasoner.dll" is a foreign library for SWI-Prolog and the "foreign_predicates.pl" is a foreign module, which interfaces the foreign library with Prolog programs. The foreign library and the foreign module are combined as the IPC extension. From the perspective of Prolog programming, the IPC extension is a set of customized predicates and from the perspective of C/C++, it is an IPC interface. The flow diagram in Fig. 3 shows how the Prolog and application processes work together bridged by IPC extension.

A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. For the IPC-based integration method, the Prolog development tool (PDT) load graph of the design pattern is illustrated in Fig. 4. Each rectangle in the figure refers to a Prolog source file. "queries.pl" is the entry of the Prolog program. For the other three rectangles, the above double lines are the list of predicates, which could be imported by other Prolog files. Lines with arrow mean dependence and the sources use predicates of the destinations. Annotation on the line means number of imported predicates. There are four Prolog source files in the figure. "queries.pl" is automatically generated at runtime by the application process. It uses many predicates in "algorithms" and one predicate in "foreign predicates", which is responsible for IPC. "Algorithms" is customizable and is the only file written before the start of the application process. It uses all the predicates in "facts" and a part of predicates in "foreign predicates", which are responsible for generating data prepared for IPC. "facts" is also generated at runtime and is only used by "algorithms".

"foreign_predicates" is the foreign module. It should also be generated at runtime, because in addition to import foreign library, it includes enumeration of all the possible facts written by the foreign predicates, which are responsible for generating data prepared for IPC.

The definition or the documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution. According to commonly accepted format used in object oriented design patterns, the definition of the Prolog design pattern is given in Table 1.

For clarification and demonstration, in the following section, the application of the Prolog design pattern is described with a case study.

3. A case study

3.1. Introduction of the target system

The target system is free software “yoyo-ontology”, which has two main functions. Firstly, it is a "yoyo-ontology" builder. This “yoyo-ontology” means a streamlined ontology, generalized from the famous gene ontology. Additionally, the software can also annotate imported datasets with nodes of a given “yoyo-ontology”. From the view of architecture, yoyo-ontology has two main modules, one for ontology building and the other for dataset annotation. The two modules are coupled loosely. The topology of a “yoyo-ontology” is directed acyclic graph (DAG), the same as gene ontology.

![Fig. 3 Flow diagram of the Prolog and application processes work together.](image-url)
An IPC-based Prolog design pattern for integrating backward chaining inference into applications or embedded systems

<table>
<thead>
<tr>
<th>Item</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern name and classification</td>
<td>IPC-based integration for backward chaining inference</td>
</tr>
<tr>
<td>Intent</td>
<td>To integrate backward chaining inference of Prolog into applications or embedded systems</td>
</tr>
<tr>
<td>Motivation</td>
<td>Prolog is powerful for AI-related problems, but is not suitable to develop many kinds of components, such as data acquisition and task scheduling. To make the best use of the advantages and bypass the disadvantages, this design pattern provides a framework to integrate Prolog into applications or embedded systems</td>
</tr>
<tr>
<td>Applicability</td>
<td>To process data with backward chaining inference and some of these data are obtained in real-time or are more difficult to be obtained by Prolog than by other methods</td>
</tr>
<tr>
<td>Structure</td>
<td>The PDT load graph of the design pattern, illustrating the necessary Prolog modules and their relationships, is shown in Fig. 4</td>
</tr>
<tr>
<td>Participant</td>
<td>Four Prolog modules, IPC extension of Prolog and application or embedded systems having IPC modules</td>
</tr>
<tr>
<td>Consequence</td>
<td>The integration is seamless and users could not be aware of the existence of the Prolog process</td>
</tr>
</tbody>
</table>

Table 1 Definition of Prolog design pattern – integrating backward chaining inference into applications or embedded systems.

Fig. 5 is a “yoyo-ontology”, namely a DAG; each node represents a concept. A node is identified as ID: XXXXX, where X is a digit. Nodes are connected with directed lines. Different heads of the lines refer to different relationships between the nodes. Here, we could ignore the type of edges. The edges point to more concrete concepts than its origins. A node list of the DAG is used to describe a record of a dataset and called an annotation. For instance, A = [ID: 10007; ID: 10009] and B = [ID: 10011] are two annotations.

For any two annotations A and B, define the semantic similarity of them as

\[
\text{Similarity}(A, B) = 1 - \frac{\text{Card}(X - Y) + \text{Card}(Y - X)}{\text{Card}(X) + \text{Card}(Y)}
\]

where X represents the union of A and all the ancestor nodes of every member of A. The relationship of Y and B is similar to X and A. Card(X) means cardinality of the set X, namely, the number of members of the set. For two sets, X and Y, X - Y means a new set, each of whose member belongs to X but not belongs to Y, namely,

\[
X - Y = \{x | x \in X, x \notin Y\}
\]

It is easy to figure out that if A = B, then the Similarity(A, B) = 1 and under other conditions,

\[
\frac{\text{Card}(X - Y) + \text{Card}(Y - X)}{\text{Card}(X) + \text{Card}(Y)} < 1
\]

So,

\[
\text{Similarity}(A, B) \in (0, 1]
\]

Now, we add another module to the software to calculate the similarity of any two given annotations. Prolog is effective to handle DAG, so the calculation of similarity resorts to Prolog. However, the data involved in the calculation must be obtained in real-time, because the topology of the ontology and the annotations are modified continuously by users according to requirements. The task of data acquisition is not suitable to Prolog and will be assigned to C++. Last but not least, the two parts should be integrated. Consider the scenario of the defined Prolog design pattern in the former subsection, it is coincident with the scenario here. In the following text, we will give the Prolog to calculate them.

3.2. Application of the design pattern

Suppose A = [ID: 10007; ID: 10009] and B = [ID: 10011], to get the similarity of A and B, we should figure out the Card(X - Y), Card(Y - X), Card(X) and Card(Y). In the following text, we will give the Prolog to calculate them.

Referring to Fig. 4, we should manage to get the four files. Coincidentally, three of them have been given in Section 2.1. For clarification, the relationships between the predicates of the three modules are illustrated in Fig. 6, in the form of PDT global view.14 In the figure, each envelope refers to a Prolog module and each rectangle in an envelope means a predicate. Lines with arrow mean dependence.

With C++, Java or other imperative languages, it is easy to generate “facts.pl” at runtime from Fig. 5. “algorithms.pl” imports the “facts” module. The fifth and sixth lines of the file define a predicate “ancestor/2” with recursion. In the following text, we will give the Prolog to calculate them.

![Fig. 5](image1.png)

Fig. 5 A DAG, representing an ontology, is composed of nodes and edges.

![Fig. 6](image2.png)

Fig. 6 Relationships between the predicates.
following text, “card X/1” and “card XY/2” are importable predicates and are responsible for calculating Card(X) and Card(X – Y), respectively. The two predicates are implemented similarly, “findall/3”, “member/2”, “append/2”, “sout/2” and “length/2” are all built in predicates. Only “foreign 01/1” is imported from foreign module “foreign predicates”.

In fact, the predicate “card X/1” is a description of Card(X) with strict constraints. If there exists a query “:- card X([ID:10007, ID:10009])”, the definition of “card X/1” will turn the problem to match a right fact defined by the predicate “foreign 01/1”. Let us look at the facts defined by “foreign 01/1” in “foreign predicates.pl”. The parameter of the predicate means numbers of nodes and the facts enumerate all the possible. The enumeration guarantees that there exists a satisfied match drive by the query mentioned above, then the match will trigger “pl_add_to_buffer()” in foreign library. Note that the number of possible nodes is determined by the topology of the DAG in Fig. 5 and the topology is changed according to its definition.

From the above description, it is clear that the inference is backward chaining and the process of the new module is in line with the flow diagram shown in Fig. 3. The design pattern presented in this paper has been adopted by yoyo-ontology 1.0.

4. Conclusions and future works

In this paper, an IPC-based method is used to integrate backward chaining inference implemented by Prolog into applications or embedded systems. A Prolog design pattern is derived from the method for reuse, whose principle and definition are provided in detail. Additionally, the design pattern is applied to a target system, which is free software, to verify its feasibility. The detailed implementation of the application is given. The implementation only uses pure C++ and SWI-Prolog and does not rely on any operating system related features, so any platforms, being compatible with SWI-Prolog, such as embedded Linux, are applicable for the method. In short, the design pattern can be further applied to wide range applications and embedded systems and the method described in this paper can also be adapted for other logic programming languages.

In our future works, we will make deeper exploration on the related topics and provide more design patterns, such as IPC-based integration for forward chaining inference.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (No.61304111), National Basic Research Program of China (No. 2014CB744904) and Fundamental Research Funds for the Central Universities of China (Nos. YWF-14-KXX-001 and YWF-13-JQCJ).

Appendix A

The software, yoyo-ontology 1.0, could be downloaded and used freely from the website http://yoyo-balance.buaa.edu.cn. All the source code files mentioned in this paper are available by email: qgong@buaa.edu.cn or keep_thinking@hotmail.com.

References


An IPC-based Prolog design pattern for integrating backward chaining inference into applications or embedded systems


Li Guoqi is a researcher and master supervisor at School of Reliability and Systems Engineering, Beihang University. He received the Ph.D. degree from the Department of Computer Science and Engineering, Shanghai Jiaotong University in 2007. His main research interests are software dependability and artificial intelligence.

Shao Yuansun is an M.S. candidate at School of Reliability and Systems Engineering, Beihang University. His area of research includes software safety and Prolog.

Hong Sheng is a researcher and master supervisor at School of Reliability and Systems Engineering, Beihang University. He received the Ph.D. degree from Beihang University. His main research interests are software reliability, network reliability and complex networks.

Liu Bin is a professor and Ph.D. supervisor at School of Reliability and Systems Engineering, Beihang University. He received the Ph.D. degree from Beihang University. His main research interests are software engineering, software testing and software dependability.