# A General Interface for Interaction of Special-Purpose Reasoners within a Modular Reasoning System

**Gleb Frank**

Knowledge Systems Laboratory, Computer Science Department
Stanford University
Stanford, CA 94305
frank@ksl.stanford.edu

**Abstract.**

The increasing significance of special-purpose reasoners in theorem proving, and the necessity for a common interaction interface for such reasoners is discussed. A candidate set of operations for such an interface is proposed. The interface is based on two important classes of objects – a reasoner and a proof step. A discussion on the importance of presenting proofs in flexible, human-readable format follows. We conclude by a general description of JTP, a model elimination theorem prover constructed using the presented interface.

## Introduction. Why Special Purpose Reasoners?

One of the most prominent tasks of computer science has always been developing efficient algorithms. A well-designed algorithm can provide an enormous speedup and in many cases can provide the only practical means of solving a problem. For example, algorithms that search for a solution in a large space are almost always dramatically slower than algorithms that derive a solution directly from logical and/or arithmetic constraints.

General-purpose theorem provers use search algorithms to produce answers to queries and proofs for these answers. While for some queries this is the best one can do, there are a lot of queries in many important logical theories whose answers can be efficiently determined by special-purpose reasoners that employ algorithms specifically designed for those queries. In these cases, it doesn't make sense to use a theorem prover's general-purpose search algorithms to find the solutions.

Some of this inefficiency is routinely overcome by hand-crafting a set of well-designed heuristics to direct the search. Such heuristics frequently work best if the knowledge base is organized in a specific way, optimized for a particular usage pattern (e.g., a limited set of queries.) Note that in these cases, the fine-tuning of a general-purpose reasoner effectively turns it into a special-purpose reasoning system.

Here are several examples of fields that would benefit from applying a special purpose reasoner.

**Evaluable predicates.** Most special purpose reasoners today are evaluable predicates. The most common example is the implementation of simple arithmetic functions. Most theorem provers also have procedures to determine the internal type of an object: variable/non-variable, whether it's a number, etc.

**Sets.** Handling sets presents difficulties for a general-purpose theorem prover. One of the reasons is that there are so many varieties of sets. Sets differ in the way membership is determined (e.g., as a list of members: {A, B, C}, or by a membership condition: integer x such that $0<x<5$). Perhaps even more important are huge variations in the size of sets. While one might want to reason about small sets by reformulating the problem in terms of the set's members, such an approach is clearly unacceptable for huge (or infinite) sets. Moreover, application of search methods to set problems often leads to combinatorial explosion of the task.

**Time logic** is another difficult topic. A lot of time interval relations are easily presentable in the form of a single predicate, *Meets* (*A*, *B*). This logical structure, while being quite elegant and absolutely correct, is at a disadvantage in a search-based theorem prover, because of the huge branching factor associated with it. Temporal relations is one of the several fields that the Cyc project (Lenat, 1995) chose to handle with a special-purpose reasoner.

**Frame systems** are generally a subset of description logics. The general-purpose reasoners typically use first-order logic to represent frame systems, with additional taxonomic inference rules added. Since the frame reasoning is a fairly narrow subset of first-

order logic, a special-purpose reasoner tuned specifically for this area would provide a substantial speedup.

**Connections to other systems** through one of the inter-agent communication protocols, like OKBC, can be considered to be special-purpose reasoners. Delegation of a goal to a remote reasoner might be risky, since latency costs can slow things down a lot. On the other hand, if the remote reasoning will proceed in parallel with local reasoning, there may not be a loss of efficiency.

**A Cache** can be viewed as a reasoner. If the cache stores *proofs* as well as the answers, then cached information can be useful even after some modifications to the underlying knowledge bases, because it is always easier to verify an existing proof by checking all the atomic elements, than to create a new one by search methods! The reasoning involved here might resemble justification techniques used in truth maintenance systems (Doyle, 1979).

## Integration of Reasoners: An Object-Oriented Approach.

To effectively use (and reuse) special-purpose reasoners and representations, one has to have an easy way to put them together. Reasoners should present a uniform interface that will allow the user to easily add, remove and replace them, to direct the flow of information between them, and utilize the results of the reasoning. To this end, reasoners should adhere to a standard as to the operations that can be performed by them.

When designing a standard reasoner interaction interface there are several important points to consider.

**Generality**. The standard interface has to be as general as possible if it is to be applied successfully to a broad field of tasks. Some of the most successful standard interfaces are extremely general, the obvious examples being, of course, TCP/IP and HTTP protocols.

**Scalability.** It is important to be able to extend an interface by adding new operations or behaviors. This feature gives a standard versatility, while preserving the generality.

**Interchangeability**. You have to be able to replace an interface with another that does the same thing in a different way.

Having considered these criteria, the thing that logically comes to mind is an **object-oriented approach**. This technique is widely used in most spheres of general software engineering; it led to development of such popular programming languages as C++ and Java, and is the basis for most modern user-interaction systems. Among the benefits of object-oriented programming are inheritance, which helps achieve scalability, and polymorphism, which is a generic tool for interchangeability.

In knowledge representation and reasoning, an object-oriented approach was taken, for example, in frame systems (Chaudhri, 1998). However, some widely-known systems prefer to view their knowledge bases as a "sea of assertions" (Cyc).

This work gives a high-level description of a candidate standard interface for reasoners and methods of their interaction based on the object-oriented approach.

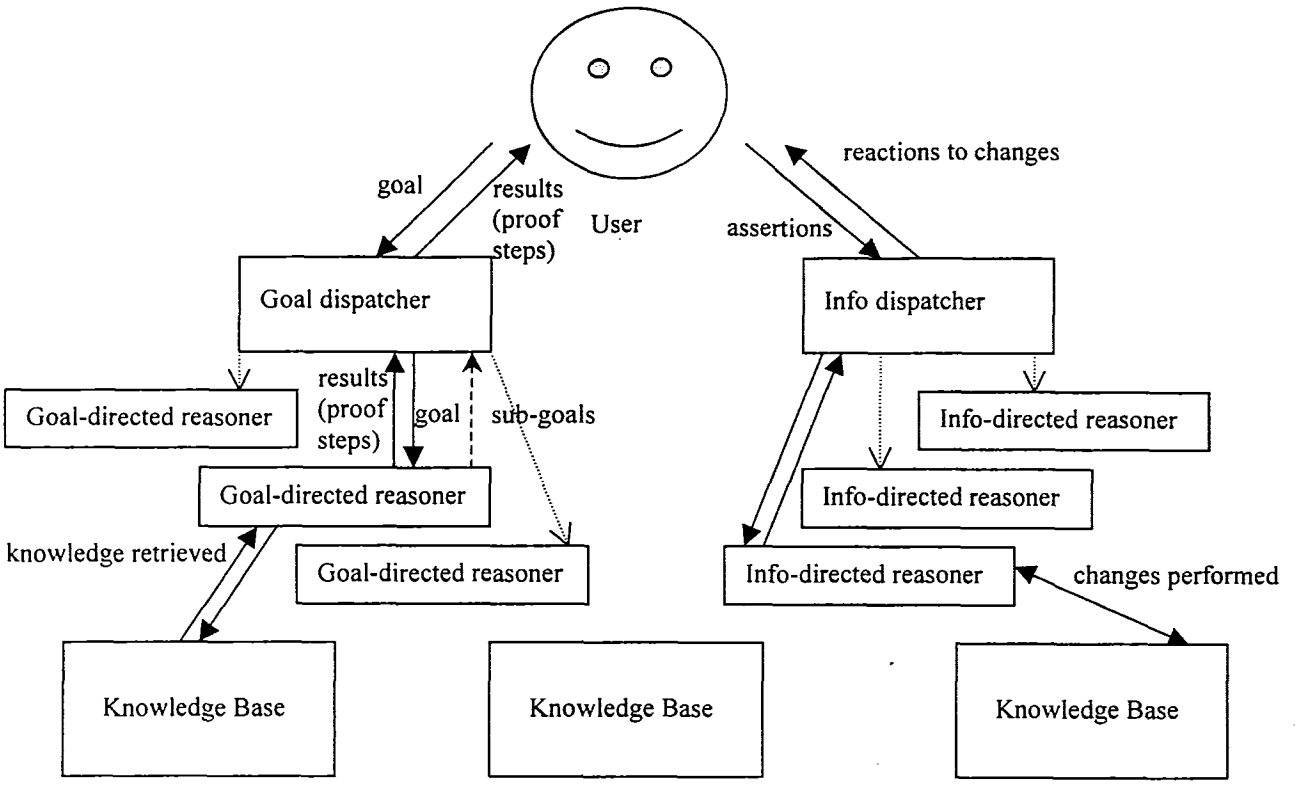
## Description of the Interface.

The two central concepts of the interface described here are those of a **reasoner** and of a **proof step**. A reasoner is a software component that transforms information, using the available knowledge to achieve goals. It is often, but not necessarily, attached to a **knowledge base**. An important sub-species of a reasoner is a **dispatcher** that directs the flow of information between other reasoners. A proof step is something that a goal-directed reasoner returns to the user containiong information about the proof for a particular goal or a description of consequences of some assertions. As opposed to architectures that typically just find a value for the user, this interface is from the very beginning oriented towards giving the user a *proof* for the solution and a human-readable *explanation.*

### Reasoner

A reasoner is an object that transforms information. In general, it serves as an intermediary between the user and a knowledge base. Reasoners can be loosely described as goal-directed or information-directed (or both). In a view/document model, a reasoner can be regarded as a view for the knowledge base, which serves as its document.

A **goal-directed** reasoner is an information source; it responds to queries (goals) from the user. For




example, a backward-chaining theorem prover receives a theorem as its query, and responds by saying whether the theorem is provable, and if so, by giving the instantiations for the variables and providing the proof.

An **information-directed** reasoner is a data receptacle. The user tells it information, which causes changes in the contents of the associated knowledge base. A forward-chaining production rule system is an example. Although strictly speaking a forward-chaining theorem prover might be goal-oriented, it behaves like an info-directed reasoner once the goal is fixed.

The user doesn't have to know about the structure or format of the knowledge base (or knowledge bases) used by a reasoner. Generally, the user interacts only with the outermost reasoners, often combined in one software component. A typical KB usually has three reasoners associated with it; one goal-directed reasoner to retrieve information from it, and two data-driven reasoners, one of which is used to tell information to the KB, the other to untell (i.e., remove) it.

A reasoner need not be connected directly to an actual knowledge base. The query (or data) can be redirected to another reasoner. In fact, such redirecting reasoners play a crucial role in the modular reasoner architecture; they serve as **dispatchers** of goals and data. Every dispatcher has a collection of reasoners it works with. Given a goal, it redirects the goal to the appropriate reasoner (or reasoners).

The simplest possible dispatcher just asks every reasoner in its command if they can process the goal. More sophisticated dispatchers use properties of reasoners to pinpoint exactly where to send the goal.

## Proof Step

A proof step is an object representing the result of an "act" of reasoning. Examples of an atomic proof step would be a single resolution in theorem proving, use of *modus ponens* in Prolog-type reasoning, or inheriting a slot value from a class in a frame system.

However, proof steps can also be compound, i.e. consisting of smaller sub-proof steps. For example, the whole proof of the top-level goal is a proof step returned by the top-level, outermost goal-directed reasoner. In a frame system, if a slot value is inherited not from a direct class, but from a superclass of a direct class, this inheritance can be

viewed as a compound proof step, consisting of two sub-steps: inheriting the superclass, and inheriting the slot value from it.

When should several proof steps be integrated into a compound proof step? It would make sense to do that when the compound proof step presents a useful abstraction of the sub-steps; something that can be, from some point of view, regarded as an atomic reasoning step. Often it is a proof for a solution of a sub-problem. The goal here is to hide the gory details of proof from those who are not interested in them, while still preserving them for the scrupulous.

A proof step is not necessarily a full proof of something. It can present a partial proof, relying on proving other goals to substantiate it. The simplest example of such partial proof step is the application of a Horn clause in Prolog, which proves the goal (a sentence unified with the head of the clause), if the sub-goals from the body of the Horn clause are proved.

(Note that after they are proved, the proof of the goal of this partial proof step is a good candidate to be made a compound proof step. The proofs of the sub-goals would be its sub-steps.)

## Operations on Proof Steps

Since the proof step is also the final result that is returned to the user. it has to be able to display itself in human-readable form. From the information displayed by the proof step, the user should be able to determine what goal this proof step proves. If this is a partial proof step. it should also have information on the sub-goals that are necessary to prove to complete it. If this is a compound proof step, one should be able to retrieve the collection of its sub-steps.

The proof step should be able to explain in detail why its goal is proved given its sub-steps (or sub-goals). For user convenience, it should provide both a short explanation (e.g. "Fido has 4 legs: value inherited from class Dog"), and a long explanation, which would describe in detail the inference rule used, possibly providing references to an even more detailed substantiation. One possible rule of thumb here is that a short explanation should fit into a line of text, while the long explanation can be something that fits into a page.

There are two obvious choices when displaying the proof step; the information can be presented either in a textual or in a graphical form. Traditional reasoners have usually displayed their result as text, which can be attributed, perhaps, to their creators' lack of interest for the HCI side of system design. However, some results can be visualized much better when shown graphically. E.g., the result of the query "What country is Paris in?" can be illustrated with the map of France with Paris on it. Since a lot (perhaps, the majority) of proof steps won't benefit from graphic representation, this operation should be optional. In other words, the default graphical representation method would be provided for textual proof steps; it would just print the explanation into the designated area.

Summarizing, a proof step has to support the following operations:

**Get Goal**: returns the goal for this proof step. The goal can be any type of object, as long as its designated goal-directed reasoner understands it.

**Get Sub-Steps**: .if this is a compound proof step, returns a collection of the sub-steps. If it is an atomic proof step, returns an empty collection.

**Get Sub-Goals**: if this is a partial proof step, returns a collection of sub-goals, otherwise an empty collection.

**Get Short Explanation**: returns a string containing a short description of the proof step. Should fit on a line of text.

**Get Long Explanation** (optional): returns a string containing the detailed explanation of the proof step. If not defined, the short explanation is used instead.

**Get Graphic Representation** (optional): returns an object able to display itself on the screen (in JTP, an instance of Java class JComponent is returned.) If not defined, the short explanation is displayed, with an opportunity to expand it to the long explanation if double-clicked.

## Operations on a Reasoner.

**process.** The principal operation a reasoner has to support is processing a goal (or a piece of data, if this is an info-directed reasoner). There are two arguments, the goal and the proof context. The goal can be any object, as long as the reasoner can handle it. If the goal is unacceptable (wrong type of object, for example), an exception is thrown.

The proof context is passed from the top-level reasoner to all other reasoners that are called. It

includes a table of parameters for reasoning, such as maximum depth of inference, whether the tracing is turned on, and if it is, what level of verbosity is required, the target for the trace messages (e.g., an output stream directed to standard output), etc. The exact set of options depends on the upstream reasoners; a reasoner that calls other reasoners can add new ones.

The method returns a collection of proof steps, representing for goal-oriented reasoners the proofs for alternative solutions for the goal and for info-oriented reasoners the consequences of the current assertion. If there are no solutions, an empty collection is returned.

There are two important special cases. The last proof step can be a *cut*. That means that the reasoner has provided all possible solutions; in other words, there are no more. As usual, the proof step contains the information necessary to substantiate this; in this case, it contains the proof of the fact that there are no more solutions. For example, if a value of a function has been calculated, there is no need to look for other values.

For an information-directed reasoner, one or more of the consequences of an assertion can be a *contradiction*. This would mean that the sentence being asserted contradicts something in the knowledge base. How exactly the contradiction is resolved is up to the particular reasoner. Usually, the offending sentence and all its consequences would be untold. But if, for example, counterfactual reasoning is being performed, the older sentence in the knowledge base might be untold instead.

**acceptable.** This operation performs a quick assessment of a goal to see if it could be handled successfully by this reasoner. It also takes a goal and the context as its arguments. It returns either *Yes*, *No* or *Maybe*.

*No* means that the goal cannot possibly be processed by this reasoner. If the goal is given to the reasoner's **process** method anyway, an exception may be thrown.

*Yes* means that the reasoner is sure it can process the goal. It doesn't mean it will find a solution, but at least it will try.

*Maybe* means that the reasoner needs more time to determine if the goal is acceptable. The caller should go ahead and **process** the goal. If the reasoner later finds the goal is bad after all, it will fail, but no exception will be thrown.

**Additionally**, a reasoner has to be able to create itself and all additional objects that it needs for operation. For example, that might include a blank knowledge base if the reasoner requires one.

### Additional Operations on Dispatchers.

A dispatcher has to support the additional functionality of organizing its reasoners. It must be able to **add** a new reasoner, or **remove** an existing one. Both operations take a reasoner as an argument, and return a Boolean value signifying the success (if true) or failure (if false) of an operation. Add fails if the reasoner is for some reason unacceptable to this dispatcher, or if it is already there. Remove fails if there is no such reasoner here. The third additional operation on dispatchers is **get Reasoners**, which returns the collection of reasoners that this dispatcher currently works with.

### Extended Reasoners: Faster Dispatch

The simple reasoner model described above has the following shortcoming: to determine which reasoners are suitable for a particular goal, the dispatcher has to contact every single one of them. This presents an unnecessary delay, especially if the system contains a lot of reasoners.

However, the object-oriented approach provides an opportunity to easily work around this. The reasoner interface can be extended to include information allowing the dispatcher to quickly direct appropriate goals here. Of course, the dispatcher has to be smart enough to take advantage of this information.

The simplest example of such an extension concerns evaluable predicates. A reasoner associated with an evaluable predicate processes only sentences with a particular relational symbol. Therefore, an additional operation is defined on these reasoners, **get Relation**. The dispatcher has a map from relation symbols to corresponding reasoners (e.g., a hash table), that it uses to quickly choose the right reasoner for every evaluable predicate.

### Another Possible Extension: Cost Assessment

Another reasonable extension of the reasoner interface would be to attempt assessing the cost of applying the reasoner to a particular goal. This can

be used to implement a more sophisticated search strategy.

### Relation with OKBC and KQML

The interface described in this paper falls into a slightly different niche from the one occupied by inter-agent communication protocols such as OKBC (Chaudhri, 1998) and KQML (Finin, 1997). Although it is not postulated formally, reasoners integrated with this architecture are supposed to typically run on the same machine, within the same address space, usually even within the same process thread. Thus the coupling of different reasoners is much tighter than is possible for a true inter-agent protocol. This allows switching of reasoners deep inside the main loop of the theorem prover, which can be prohibitively expensive if the switch requires a network access, or even a process context switch.

## JTP: a Plug-and-Play Modular Theorem Prover

JTP is a theorem prover that attempts to take advantage of the modular architecture described above. This is a work in progress, and not all of the component reasoners are implemented yet. At the core of JTP is a model elimination backward chaining theorem prover, which initially started out as a Java port of KSL's ATP theorem prover, written by Adam Farquhar. JTP is also an OKBC server.

JTP is implemented in Java. This choice was largely determined by the object-oriented nature of the reasoner interaction architecture. The structure of Java programs is beneficial for the plug-and-play organization; the code for a particular reasoner doesn't even have to be recompiled to include it into the system. The high portability of Java classes is an additional benefit.

### Reasoners within JTP

**Main dispatcher** redirects goals to all the other reasoners. This is the place to plug in arbitrary external reasoners. This reasoner also directs the iterative-deepening search for the solution.

**Evaluable predicate dispatcher** checks if the goal is an evaluable predicate identifiable by its relation, and if so, sends it to the appropriate reasoner.

**Evaluable predicate reasoners** are used to compute evaluable predicates.

**Frame reasoners:** The special-purpose frame KB has a query reasoner, a telling reasoner and an untelling reasoner for a simple monotonic frame language that is a subset of the OKBC knowledge model. Both data-driven reasoners perform considerable forward-chaining inference (a subset of description logic reasoning).

**Prolog-type reasoner** does the usual, general-purpose model elimination reasoning, returning partial proof steps corresponding to clauses being used.

**Linear reduction reasoner** uses the goal ancestor KB maintained by the main dispatcher to perform linear reduction of goals.

**Cache** is used to save certain recent proofs for later use.

JTP is still "under construction", but due to the extendable nature of the architecture, it is already fully functional. We currently experiment on using JTP as a module in development of educational software.

## Summary.

We have discussed the need for the special-purpose reasoners in modern theorem proving, and the need for a general, extendable standard interface for such reasoners. We have provided a candidate interface based on the object-oriented approach. This interface is the basis for JTP, a plug-and-play modular theorem prover implemented in Java..

## References.


Chaudhri, V. et al. OKBC: A Programmatic Foundation for Knowledge Base Interoperability. *Proc. Nat'l Conf. Artificial Intelligence/AAAI-98.* AAAI Press, Menlo Park, Calif., 1998.

Cyc Knowledge Server *description at* http://www.cyc.com/products2.html

Doyle, J. A truth maintenance system. *Artificial Intelligence*, 12(3), 1979.

Finin, T., Labrou, Y., and Mayfield, J. KQML as an agent communication language. In *Software Agents*, Jeff Bradshaw (Ed.), MIT Press, Cambridge, 1997.

Lenat, D.B. Cyc: A Large-Scale Investment in Knowledge Infrastructure. *Communications of the ACM* 38, no. 11., 1995.