JTP: A System Architecture and Component Library for Hybrid Reasoning

Richard Fikes Gleb Frank Jessica Jenkins

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

Abstract

We describe an object-oriented modular architecture for hybrid reasoning (called the JTP architecture), a library of general-purpose reasoning system components (called the JTP library) that supports rapid development of reasoners and reasoning systems using the JTP architecture, and a case study of a temporal reasoner that was incorporated into an existing multi-use reasoning system employing the JTP architecture and library.

1. Introduction

In this paper, we describe an object-oriented modular architecture for hybrid reasoning (called the JTP architecture), a library of general-purpose reasoning system components (called the JTP library) that supports rapid development of reasoners and reasoning systems using the JTP architecture, and a case study of a special-purpose temporal reasoner that was incorporated into a multi-use reasoning system (called the JTP system) employing the JTP architecture and library.

The research reported on in this paper addresses the challenges inherent in developing systems that can perform effective automated reasoning for multiple reasoning tasks using large quantities of heterogeneous knowledge. Such systems are needed in many application areas, including intelligence analysis, autonomous control of complex systems, and question-answering from the Semantic Web. Experience with automated reasoners has made increasingly clear that in order to effectively deal with the demands of such complex "real world" reasoning problems, general-purpose reasoners need to be augmented with special-purpose reasoners that embody both domain-specific and task-specific expertise. That is, such problems require *hybrid reasoning*. (Other examples of hybrid reasoners include [Brachman *et al* 85], [Myers 94], and [Stickel 85].)

The JTP hybrid reasoning system architecture and reasoning system component library is intended to enable the rapid building, specializing, and extending of hybrid reasoning systems. Each reasoner in a JTP hybrid reasoning system can embody special-purpose algorithms that reason more efficiently about particular commonly-occurring kinds of information. In addition, each reasoner can store and maintain some of the system's knowledge using its own specialized representations that support faster inference about the particular kinds of information for which it is specialized.

2. JTP System Architecture

The JTP architecture is organized into layers of abstraction. At the most abstract layer, very few commitments are made about the representation language and the nature of the content the system is working with. The idea is to give the implementer maximum flexibility when creating reasoning systems within the architecture while still providing him with the structure and common vocabulary to facilitate integration and interaction of heterogeneous modules. Nevertheless, all the existing reasoning systems built using the JTP architecture share certain common features, and while the architecture does not impose them, we will limit ourselves to discussing the kinds of systems with which we have the most experience.

At the top level of abstraction, the JTP architecture is representation language independent. For the purposes of this paper, we are assuming that the representation language is a symbolic logic language, and the implemented JTP reasoning system uses a first-order logic (FOL) representation language (i.e., KIF 3.0 [Genesereth and Fikes 92]). We consider the case when there is a single knowledge base (KB) that is a logical theory with respect to which all processing is done. A KB contains a set of symbolic logic sentences; for each such sentence it can provide a justification in the form of a *reasoning step*. A reasoning step is a central concept in this architecture.

The primary work of the system is performed by modules called *reasoners*. There are *telling*

reasoners that are invoked when a sentence is being added to the KB and *asking reasoners* that are invoked when the KB is being queried. Reasoners produce reasoning steps, each of which is a partial or completed proof.

A reasoning system using the JTP architecture needs some means of determining to which of its arbitrary number of reasoners to route its inputs. That capability is provided by reasoners in the system that act as *dispatchers* of an input to other reasoners that the dispatcher determines may be able to process the input. Each dispatcher has a set of child reasoners associated with it and serves as a transparent proxy for those child reasoners.

2.1. Reasoning Steps

A reasoning step represents a partial or completed proof. It consists of the following elements:
a *claim***,** which is a sentence; this is the sentence that this reasoning step justifies;

1) a set of *premises* the justification relies upon, each of which is a sentence;
2) a set of *child reasoning steps*; this justification relies on their claims;
3) a set of *variable bindings*;
4) a set of *removed assumptions*, each of which is a sentence. Each of these can be used by child resoning steps without making this reasoning step conditional on that assumption.
5) an *atomic justification* that justifies the claim of this reasoning step given its premises and claims of its children. The atomic justification can be either "Axiom", "Assumption", or an inference rule. Reasoning steps justified as "Axiom" or "Assumption" do not have any premises, children or removed assumptions; an inference rule serves to infer the reasoning step's claim from its premises and the claims of the child reasoning steps, and also to justify the removal of assumptions.

Note that wherever a sentence is used in the reasoning step, one can instead use a *sentence schema*, i.e., a sentence with some free variables in it. The reasoning step then represents a proof schema.

2.1.1. Proofs

A reasoning step is conditional on assumption A if (1) its claim is A and is justified as "Assumption" or (2) one of its children is conditional on A and A is not on its list of removed assumptions. A *proof* is a reasoning step that (1) has no premises, (2) has no descendant reasoning steps that have any premises, and (3) is not conditional on any assumptions. Note that any reasoning step whose atomic justification is "Axiom" is a proof.

A proof P is said to be a *completion of a reasoning step* RS when P and RS have the same claim schema and the same set of removed assumptions, the variable binding set of P is a superset of the variable binding set of RS, and every child reasoning step of P is either a child reasoning step of RS, or has as a claim schema a premise schema of RS, or is a completion of a child reasoning step of RS.

2.2. Reasoners

Reasoners do the actual reasoning work by producing reasoning steps. An output of a reasoner is a sequence of reasoning steps that are accessed through a handle called an enumerator. Every time an enumerator is accessed, it returns the next reasoning step in the sequence or indicates that there are no more. Note that some reasoners produce all the reasoning steps in a sequence at once while others produce reasoning steps one at a time.

2.2.1. Telling Reasoners

A telling reasoner takes as input a proof. The proof may represent either a sentence that is being told to the system (justified as an "Axiom or an "Assumption"), or a result of a previous inference (justified by an inference rule). The reasoner may assert the sentence to one or more knowledge stores, produce additional inferences in the form of new proofs, or signal that a contradiction has been found. The reasoning steps produced by telling reasoners are completed proofs of additional sentences that are inferred from the input proof and the sentences in the KB.

2.2.2. Asking Reasoners

An asking reasoner accepts as input a sentence schema S of which the system is attempting to prove instances. The reasoner attempts to produce reasoning steps with S as the claim schema.

2.2.3. Dispatchers

A dispatcher is a special type of reasoner that does not actually do any reasoning itself but serves as a transparent proxy for other reasoners. Every dispatcher has a set of child reasoners associated with it. The principal purpose of each dispatcher is to route inputs to its child reasoners. Specifically, when a dispatcher is called, it identifies the subset of its child reasoners that are to handle the input. Each identified child reasoner processes the input and produces an enumerator of reasoning steps. The dispatcher concatenates these enumerators into one enumerator and returns it as its output.

There are both telling and asking dispatchers. Note that although a dispatcher does not do any reasoning itself, if we consider a dispatcher with its set of children and view it as a black box, it

behaves exactly as a reasoner of corresponding type (telling or asking). In particular, a dispatcher can be used as a child reasoner for another dispatcher. This allows one to build complex hierarchical trees of reasoners.

2.3. Knowledge Stores

The manner in which a reasoner stores sentences and justifications (or whether it stores them at all) is not constrained by the JTP architecture. A knowledge store can be loosely defined as a pair of reasoners, one asking and one telling, that serve as access ports to the sentences in the store. The telling reasoner is used to add sentences to the store; the asking reasoner is used to query the store.

2.4. Reasoning Context

A reasoning context serves as the interface to JTP. It consists of two control reasoners -- one asking and one telling -- and two control dispatchers -- one asking and one telling. These dispatchers serve as roots of hierarchies of reasoners, some of which serve as access ports for knowledge stores. The reasoning context supports a set of commands including `Tell`, `Ask`, and `Load KB`.

2.4.1. Control Reasoners and Dispatchers

2.4.1.1. Telling Control Reasoner and Dispatcher

There is a distinguished telling reasoner in the system called the *telling control reasoner* (TCR) that manages the processing that occurs when a sentence is told to the KB. The `Tell` command calls the TCR to process a sentence being told as an axiom to the KB, and reasoners are expected to call the TCR to add sentences to the KB that they have derived.

The TCR initiates the processing of the claim C of its input proof by dispatching the proof to a distinguished telling dispatcher called the *telling control dispatcher* (TCD). The TCD controls access to the hierarchy of telling reasoners. In an iterative process, the TCR gets sentences derived from C by pulsing the enumerator returned by the TCD. The TCR passes these derived proofs to the TCD to derive further proofs, and so on.

2.4.1.2. Asking Control Reasoner and Dispatcher

There is a distinguished asking reasoner in the system called the *asking control reasoner* (ACR) that manages the processing that occurs when the system is attempting to prove instances of a given sentence schema. The ACR attempts to produce proofs of its input sentence schema S by: (1) calling a distinguished asking dispatcher called the *asking control dispatcher* (ACD) with S as the input; (2) pulsing the enumerator produced by the call to ACD to obtain reasoning steps RSi with S as the claim schema; and (3) attempting to produce proofs Pij that are completions of each reasoning step RSi. While attempting to produce proofs of its input S, the ACR also maintains the set of assumption schemas that are available for use in proofs so that if a subgoal SG that the ACR is trying to prove while completing a reasoning step is an instance of an available assumption schema, then it can create a proof of SG with justification "Assumption".

Producing a proof P that is a completion of a reasoning step RS having bindings B and assumption schemas AS1,…,ASn involves producing proofs for instances of the premise schemas of RS given any assumptions that are instances of AS1/B,…,ASn/B and incorporating those proofs as child reasoning steps and additional bindings into P. Each such attempt to produce proofs for instances of a premise schema PR/B is done by adding AS1/B,…,ASn/B to the set of available assumption schemas and either using an instance of an available assumption schema as a proof justification for PR/B or calling ACD with PR/B as the input as if the ACR were being called recursively.

2.4.2. Commands

2.4.2.1. Tell

The `Tell` command takes as input a sentence S and adds it to the KB. The command does this by forming a proof P having claim S and atomic justification "Axiom" and calling the TCR with P as input. If the telling of S results in the derivation of an inconsistency, then `Tell` returns a proof of the inconsistency. Otherwise, `Tell` returns the enumerator returned by the TCR. The output of that enumerator when pulsed is a proof produced by a telling reasoner for a sentence that was derived and added to the KB during the telling of S.

2.4.2.2. Ask

The `Ask` command takes as input a sentence schema S, and produces as output an enumerator whose output when pulsed is an unconditional proof of an instance of S (i.e., a proof having S as the claim schema). `Ask` produces its output by calling the ACR with S as input.

2.4.2.3. Load KB

The `Load KB` command takes as input a text file F and attempts to produce a KB from F. Parsing the contents of F produces either a collection of sentences or syntax errors. If no syntax errors are found, each sentence is told to the system using the `Tell` command. The sequence of `Tell` commands

is halted if a `Tell` command reports that a logical inconsistency has been found.

3. JTP Library of Reasoning System Components

3.1. Evaluable Relation Reasoners

The library includes a collection of asking reasoners that provide standard functionality for evaluable relations such as arithmetic operators. Each of these reasoners processes input literal schemas that have a specific relation symbol. When given such a literal with constants as arguments, the reasoner can calculate whether the literal's arguments belong to the relation. Most such reasoners also can determine a true instance of the input if one of the arguments is a variable; in particular, reasoners whose relation is a function can calculate the function's value (i.e., the last argument of the literal) from the function's arguments (all other arguments of the literal.)

3.2. Horn Clause Telling Reasoner Generator

The library contains a module that takes as input a file of multi-literal Horn clauses and produces as output for each Horn clause HCi a Horn clause telling reasoner (HCTR) that uses HCi as a forward-chaining rule and conceptually works as follows. If HCi is of the form "(=> A C)", where A and C are literals, then the reasoner will respond to the telling of any instance A/B of A by storing C/B in the KB and enumerating a proof of that stored sentence. If HCi is of the form "(=> (and A1 … An) C)", where each of A1, …, An, and C is a literal, then the reasoner will respond to the telling of any instance Aj/B of any antecedent Aj of HC by creating and adding to the system a HCTR for Horn clause HCij defined as follows. If n=2 (i.e., HCi has exactly two antecedents) and Am denotes the unmatched antecedent of HCi, then HCij is "(=> Am/B C/B)". Otherwise, HCij is "((=> (and A1/B … Ai-1/B Ai+1/B … An/B)".

3.3. Model Elimination Asking Control Reasoner

The library includes an asking control reasoner that is a model elimination theorem prover [Stickel 88] modified to function in the JTP architecture. In particular, the model elimination subgoaling and search mechanism has been modified to work with the architecture's reasoning steps and to call the ACD in order to attempt to prove each subgoal.

3.4. Generalized Modus Ponens Knowledge Store and Reasoners

The library contains a knowledge store for arbitrary FOL sentences and a telling reasoner and an asking reasoner that use the knowledge store. The telling reasoner adds FOL sentences to the knowledge store in the form of clauses. The asking reasoner processes input goals that are literal schemas, and it tries to unify the goal with the literals belonging to clauses being kept in the store. If successful, it returns reasoning steps that are partial proofs of instantiations of the goal with the help of these clauses.

For example, assume the following axioms are told:

```
(forall (?f)
  (=> (exists (?y) (father ?f ?y)))
      (male ?f))
(forall (?x)
  (=> (animal ?x)
      (<=> (male ?x) (~(female ?x)))))
```

After conversion to clause normal form, the KB will contain the following clauses:

```
(or (male ?f) (~(father ?f ?y)))
(or (male ?x) (female ?x) (~(animal ?x)))
(or (~(male ?x))
    (~(female ?x)) (~(animal ?x)))
```

If the query is `(male joe)`, the asking reasoner will produce the following two reasoning steps, each of which will claim `(male joe)`: (1) a reasoning step with a premise `(father joe ?y)` and a binding of `?f` to `joe`; and (2) a reasoning step with premise `(~(female joe))`, premise `(animal joe)`, and a binding of `?x` to `joe`.

3.5. Dispatchers

The library includes the following two dispatchers: a *Sequential Dispatcher* that simply sends each input to all its children in sequence, and a *Relation-Based Dispatcher* (RBD). An RBD indexes its child reasoners on relation symbols and optionally on arity and polarity. It dispatches inputs that are literals to the reasoners that are indexed under the literal's relation symbol; it does nothing for inputs that are not literals. If a child reasoner of an RBD specifies a positive integer N arity, then only literals having N arguments are sent to that reasoner. Similarly, if a child reasoner of an RBD specifies a polarity of either "negative" (or "positive"), then only literals that are (not) negated are sent to that reasoner.

4. A Case Study Using the Architecture

We have built multiple reasoners for our JTP system and have found the JTP architecture and library to be extremely effective at enabling new reasoners to be rapidly implemented and incorporated into the system. We present here an example of such reasoner development to illustrate the use and value of the architecture and library.

We recently expanded our JTP reasoning system

by developing a capability for reasoning with time-dependent knowledge. We accomplished this by first developing a Reusable Time ontology [Fikes and Zhou 02] that provided definitions for time points, time intervals, temporal relations on time points such as `before` and `after`, the Allen relations on time intervals [Allen 83], calendar objects such as the months of the year, etc. We then developed domain-specific telling and asking reasoners for the temporal relations on time points and the Allen relations on time intervals that made use of a special-purpose knowledge store called a Temporally Labeled Graph (TLG) as described in [Delgrande *et al* 2001].

We describe the development of those temporal reasoners in this section.

4.1. Temporally Labeled Graph

We began by implementing a TLG. A TLG provides data structures for storing partial ordering relationships among time points and the following set of commands for querying, updating, and retrieving data from the data structures:

`query` – The `query` command takes as inputs two time points, A and B. The command returns the relation between A and B, which can be `before`, `after`, `equal-point`, `before-or-equal`, `after-or-equal`, `not-equal`, or `any-relation`.

`update` – The `update` command takes as inputs two time points and the relation between them. The TLG is updated appropriately. This command has no return value.

`getAllPointsAfter`, `getAllPointsBefore`, `getAllPointsEqual`, `getAllPointsAfterOrEqual`, `getAllPointsBeforeOrEqual` – These commands take as input a single time point A and return a collection of all known time points that occur in the respective relationship (i.e., after, before, same time as, after or at the same time as, or before or at the same time as) with A.

While this TLG and its commands allow the user to manipulate time points in many desirable ways, it does not allow for the assertion and querying of time intervals. It also is not integrated within a larger system that allows a user to have a KB of which time points are only one component. The JTP architecture allows us to integrate this TLG into a confederation of reasoners so that it can be more generally useful in reasoning tasks.

4.2. The TLG as a Knowledge Store

The first step in the integration was to provide wrappers for the TLG to conform to the JTP architecture's requirements for a knowledge store. The TLG provides a knowledge store for literals whose relation is any of `before`, `after`, `equal-point`, `before-or-equal`, `after-or-equal`, or `not-equal`, and whose arguments are constants representing time points.

The TLG was augmented to support associating a set of justifications with each asserted relationship between time points, and new commands were implemented for adding and retrieving justifications. The TLG's `update` command was used to implement the `add sentence` command and the TLG's `query` and `getAllPoints---` commands were used to implement the `retrieve sentences` command.

4.3. Reasoners for Time Points

We then implemented reasoners for relations between time points as follows.

4.3.1. Time Point Telling Reasoner

This reasoner processes proofs being told to the KB whose claim is a literal whose relation is any of `before`, `after`, `equal-point`, `before-or-equal`, `after-or-equal`, or `not-equal`, and whose arguments are constants representing time points. The reasoner simply stores the proof and its claim in the TLG knowledge store using its `add sentence` command.

4.3.2. Time Point Asking Reasoner

This reasoner processes literal schemas whose relation is `before`, `after`, `equal-point`, `before-or-equal`, or `after-or-equal`, or literal schemas whose relation is a variable and whose arguments are constants representing time points. The reasoner uses the `retrieve sentences` command of the TLG knowledge store to determine answers to the query. The reasoner enumerates a reasoning step for each answer produced by the retrieve command containing appropriate bindings for the variables in the input literal schema.

4.4. Time Intervals

To provide reasoners for the Allen relations on time intervals, we used the functions `starting-point` and `ending-point` defined in the Reusable Time ontology.

4.4.1. Telling

When a sentence using an interval relation is told, the interval relation can be translated into time point relations between the start and end points of the two intervals using axioms from the ontology such as the following:

```
(=>  (precedes ?t1 ?t2)
     (before  (ending-point ?t1)
              (starting-point ?t2)))
```

We used the Horn clause telling reasoner generator from the JTP library to produce a telling reasoner for each such Horn clause from the ontology. These telling reasoners derive literals relating the

start and end points of intervals that can be stored in the TLG knowledge store by the Time Point Telling Reasoner.

For example, if the sentence `(precedes Joe-Birthday Amy-Birthday)` is told, then the sentence `(before (ending-point Joe-Birthday) (starting-point Amy-Birthday))` will be derived and added to the TLG knowledge store.

Note that in order to create these telling reasoners, we only had to expend the minimal effort of designing the Horn clauses that the Horn Clause Telling Reasoner Generator used to produce its HCTRs.

4.4.2. Asking

While the translation of interval relations into point relations allows JTP to successfully process queries about time points, including the ones that have been implicitly defined via intervals, we would also like for the system to answer queries about intervals.

The JTP library includes a Generalized Modus Ponens asking reasoner. Adding to the KB sentences from the ontology that define the point relations in terms of interval relation allows that reasoner to reformulate queries about interval relations in terms of point relations, and the Time Point Asking Reasoner is then able to complete the query using the TLG knowledge store.

As an example, the following Horn clause axiom is in the ontology:

```
(=>   (before  (ending-point ?t1)
               (starting-point ?t2))
      (precedes ?t1 ?t2))
```

The point relation literals (`before (ending-point Bob-Vacation) (starting-point Jill-Vacation))` and `(after (starting-point Betty-Vacation) (starting-point Jill-Vacation))` are also in the KB. If the user asks the query `(precedes Bob-Vacation Betty-Vacation)`, the Generalized Modus Ponens reasoner will unify the sentence with the clause above, and JTP will attempt to solve the goal `(before (ending-point Bob-Vacation) (starting-point Betty-Vacation))`. The Time Point Telling Reasoner will then successfully be able to answer this query with the use of the TLG knowledge store.

4.5. Integrating Temporal Reasoning

Configuring a system that uses the JTP architecture to include temporal reasoning is as simple as adding the Time Point Telling Reasoner and the Horn Clause Telling Reasoner Generator to the TCD and adding the Time Point Asking Reasoner and the Generalized Modus Ponens reasoners to the ACD. The necessary sentences must also be told to the Generalized Modus Ponens knowledge stores through the reasoning context's `Tell` command.

If the KB that the user wishes to work with requires the use of other special-purpose reasoners, these can also be added to the dispatchers. JTP's architecture allows users to effortlessly combine different special-purpose reasoners for use with complex KBs.

References

[Allen 83] James Allen; "Maintaining Knowledge about Temporal Intervals"; Communications of the ACM, 26(11), pp.832-843; November 1983

[Brachman *et al* 85] Ronald Brachman, Victoria Gilbert, and Hector Levesque; "An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of KRYPTON"; IJCAI-85; 1985; pp 532-539.

[Delgrande *et al* 2001] James Delgrande, Arvind Gupta, and Tim Van Allen; "A Comparison of Point-Based Approaches to Qualitative Temporal Reasoning"; Artificial Intelligence Journal, 131, 1-2, 2001; pp. 135-170.

[Fikes and Zhou 02] Richard Fikes and Qing Zhou; "A Reusable Time Ontology"; AAAI-2002 Workshop on Ontologies and the Semantic Web; Edmonton, Canada; July 29, 2002.

[Genesereth and Fikes 92] Michael Genesereth and Richard Fikes; "Knowledge Interchange Format, Version 3.0 Reference Manual"; KSL Technical Report 92-86; Knowledge Systems Laboratory, Computer Science Department, Stanford University; Stanford, CA; 1992.

[Myers 94] Karen Myers; "Hybrid Reasoning Using Universal Attachment"; AI 67 (1994); pp 329-375.

[Stickel 85] Mark Stickel; "Automated deduction by theory resolution"; IJCAI-85; 1985; pp 455-458.

[Stickel 88] Mark Stickel; "A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler"; Journal of Automated Reasoning; Vol. 4; pp
353-380; 1988.