# Joshua*: Uniform Access to Heterogeneous Knowledge Structures or Why Joshing is Better than Conniving or Planning

Steve Rowley, Howard Shrobe, Robert Cassels
Symbolics Cambridge Research Center
11 Cambridge Center, Cambridge, MA 02142
Walter Hamscher
MIT Artificial Intelligence Laboratory
545 Technology Square, Cambridge, MA 02139
Howard Shrobe is also a Principal Research Scientist at the MIT Artificial Intelligence Laboratory.

**Abstract**

This paper presents *Joshua*, a system which provides syntactically uniform access to heterogeneously implemented knowledge bases. Its power comes from the observation that there is a *Protocol of Inference* consisting of a small set of abstract actions, each of which can be implemented in many ways. We use the object-oriented programming facilities of Flavors to control the choice of implementation. A statement is an instance of a class identified with its predicate. The steps of the protocol are implemented by methods inherited from the classes. Inheritance of protocol methods is a compile-time operation, leading to very fine-grained control with little run-time cost.

Joshua has two major advantages: First, a Joshua programmer can easily change his program to use more efficient data structures *without changing the rule set* or other knowledge-level structures. We show how we thus sped up one application by a factor of 3. Second, it is straightforward to build an interface which incorporates an existing tool into Joshua, *without modifying the tool.* We show how a different TMS, implemented for another system, was thus interfaced to Joshua.

## 1. The Quandary

Advances in computer science are often consolidated as programming systems which raise the abstraction level and the vocabulary for expressing solutions to new problems. We have seen little permanent consolidation of this form in AI.

We believe that there are four causes for the brief tenure of AI programming systems:

1. Some are overly restrictive in their choices of paradigms, data structures and representations.
2. Others provide little guidance in how to usefully employ the grab-bag of tools in the system.
3. Virtually all erect a syntactic barrier between the AI system and its surrounding procedural framework (e.g., Lisp).
4. Finally, it is very difficult to incorporate existing facilities which were not coded within the framework.

These all result from the tension between the expressiveness of a problem solving language and the flexibility and efficiency of its implementation. Fully expressive languages, such as the Predicate Calculus, are invaluable because they provide a uniform framework within which one can capture all aspects of a problem's solution. Historically, the expressiveness of such languages has forced implementators to employ uniform algorithms and data structures capable of supporting their generality. As a consequence it has been difficult to incorporate an external system which uses different data representations, such as a relational database, without special purpose kludgery. Furthermore, each such system requires different kludgery. One of our goals is to provide a framework for incorporating such systems systematically.

In addition, many problem domains don't require all the expressive power of a general purpose language. In such cases, implementors have been able to exploit the limited expressiveness of a domain to build a highly efficient special purpose problem solving language. Such a language cannot, in principle, support general problem solving, but where applicable it is highly desirable. A common example of this is when we are dealing exclusively with triples of objects, attributes and values. The popularity of frame-like languages is accounted for by the fact they are very efficient in this limited domain, even though they are incapable of dealing with full quantification. A frame language can very efficiently reason about the properties of Opus and birds in general, although it cannot even express a statement like "everybody like something, but nobody doesn't like Opus". If all one wants to do is the former kind of reasoning, then a frame language provides a reasonable tradeoff.

Another example is reasoning about physical artifacts where again the full expressive power of PC is unnecessary. Indeed, it is much more natural and much more efficient to build a representation which emphasizes the objects and their connectivity, mirroring the topology of the artifact in the topology of the data structures. The constraint language of [Sussman and Steele] is an example of such a specialized language.

The second of our goals is to be able to exploit many of these specialized techniques in a single system without closing ourselves off from the use of a problem solving language of full expressive power. In this paper we will illustrate this quandary and Joshua's solution to it using the task of building a trouble-shooter for the digital circuit shown in Figure 1; the trouble-shooter will be one very similar to those in the literature (e.g. [Davis, et al.]), our goal is to illustrate Joshua's capabilities, not to discuss trouble-shooting. First we will show how the default Joshua facilities can be used to solve this problem, producing a solution of reasonable efficiency. Then we will show how we can include a specialized constraint language for this problem within the broader Joshua framework. This will lead to a dramatic improvement in performance, even though *it will leave unchanged all of the knowledge level structures of the original solution.*

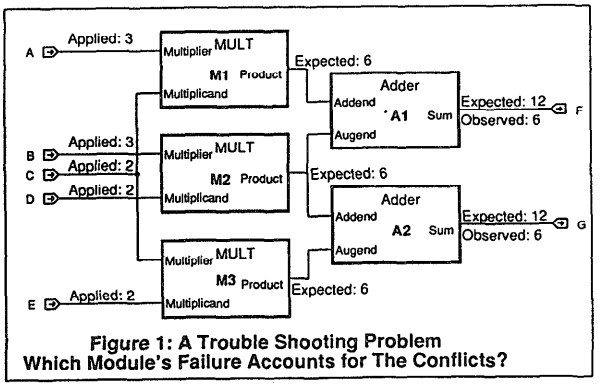


**Figure 1: A Trouble Shooting Problem**
**Which Module's Failure Accounts for The Conflicts?**

*"And Joshua burnt Ai, and made it an heap forever, even a desolation unto this day." -- Joshua 8:28, KJV

## 2. Our Solution: Uniformly Accessible Heterogeneous Representations

Joshua solves the problem using the abstraction power of the Lisp programming environment. In particular, it relies on the object-oriented facilities of Flavors, although the facilities of the emerging Common Lisp object-oriented programming standard would serve as well. The main features of Joshua are:

- There is a **Uniformly Accessible but Heterogeneous Data Base of Statements.** Two Lisp forms, ASK and TELL, provide the interface to this database. ASK queries the data base, finding facts which explicitly match the query as well as those implied by backward chaining rules and other inferential capabilities. TELL inserts a statement and computes its consequences by invoking forward chaining rules and other inferential capabilities. ASK and TELL may implement their behavior in any manner desired and the statements may be represented using a variety of different data structures. The contract of ASK and TELL is functional, not implementational.
- There is a **Fine-Grained Protocol of Inference** in which each distinct step of processing a statement is identified and made accessible. This protocol is hierarchical as well as fine-grained. ASK, TELL, rule compilation, rule triggering and truth maintenance are all parts of this protocol *as are their component steps.* To use a novel set of data structures, for example, one needs to change only a few, small steps of the protocol.
- **Each step of the protocol is a** *Generic Function*, i.e., an abstract procedure whose concrete implementation is found by dispatching on the data types of its arguments. The generic functions are implemented using object-oriented programming techniques (in particular Flavors). Statements are regarded as *instances*. Predicates are identified with the *classes*. The protocol steps are implemented as *methods*.
- **There is a modular inheritance scheme which allows facilities to be identified and reused.** The classes corresonding to predicates are the leaves of an inheritance lattice. A more abstract class in this lattice is thought of as a *model* for implementing part or all of the protocol, supplying *Methods* only for those few protocol steps that it handles in a unique way. Inheritance of methods happen at compile time; there is no run-time cost.
- There is a **well-crafted default implementation** of each step of the protocol provided in the system. However, the protocol is hierarchical, so modifications can be focused on lower level protocol steps, preserving the gross structure. Most models continue to use most of the default methods, thus satisfying the *Principle of Incrementality* that the effort required to effect a modification of behavior should be proportional to the size of the changed behavior.

These features allow Joshua to incorporate outside tools easily and use specialized representations where desirable. In the rest of this paper we will illustrate these points. First, we will show a straightforward Joshua implementation of a digital trouble-shooter which is reasonably efficient. However, a solution using a constraint language approach would greatly improve the efficiency. To see how such a representation can be incorporated, we will present the Protocol of Inference in some detail. Then we will show that to incorporate this alternative implementation we will only have to provide a few protocol methods. We will modify no knowledge-level structures of our original implementation. Finally, we will present a brief example of how Joshua can incorporate tools built outside the Joshua framework.

## 3. Basic Joshua

Joshua's syntax is uniform and statement-oriented; statements are delimited by brackets and variables are indicated by a leading equivalence sign. Free variables are, as usual, universally quantified. The core of Joshua is provided by the two generic functions TELL and ASK. TELL adds a statement to the data base of known facts and then performs whatever antecedent inferences are possible. ASK takes two arguments, the first of which is the *query*; the second argument, called the *continuation*, is a function which is called in a binding context created by unifying the query and matching statements. The continuation is called once for each statement satisfying the query whether this statement is explicitly present or is deduced. Many of Joshua's deductive capabilities are built using forward and backward-chaining rules. A Truth Maintenance System provides the ability to make and retract assumptions, to explain the reason for believing any statement, or to find the set of statements supporting any conclusion in the database.

Figure 2 shows how one would use Joshua to build a hardware trouble-shooting system similar to those in [Davis et al.], [Genesereth] or [deKleer & Williams]. A simulator for the circuit is built by defining rules which describe adders, multipliers, and wires and then by executing a Lisp procedure which TELLs what components are present and how they are connected. A simulation is run by providing initial values for the inputs of the circuit. A backward-chaining rule captures the notion of a *conflict*, a point in the circuit at which the predicted and observed values disagree. The trouble-shooter's goal is to find all modules in the circuit whose failure could plausibly account for each conflict. This is done in the procedure FIND-CANDIDATES which uses the TMS to find the intersection of the sets of assumptions supporting each conflict.

```
(DEFRULE ADDER-FORWARD (:FORWARD :IMPORTANCE 1)
  ;; Compute adder output from inputs
  IF [AND [TYPE-OF ≡A ADDER-BOX]
          [STATUS-OF ≡A WORKING]
          [VALUE-OF INPUT A ≡A ≡V1]
          [VALUE-OF INPUT B ≡A ≡V2]]
  THEN (TELL `[VALUE-OF OUTPUT SUM ,≡A ,(+ ≡V1 ≡V2)]))

(DEFRULE MULTIPLIER-INFERENCE (:FORWARD)
  ;; Compute multiplicand from product
  ;; and multiplier by dividing
  IF [AND [TYPE-OF ≡M MULTIPLIER]
          [STATUS-OF ≡M WORKING]
          [VALUE-OF PRODUCT ≡M ≡V1]
          [VALUE-OF MULTIPLIER ≡M ≡V2]]
  THEN (UNLESS (= 0 ≡V2)
         (TELL `[VALUE-OF MULTIPLICAND ,≡M ,(/ ≡V1 ≡V2)])))

(DEFRULE WIRE (:FORWARD :IMPORTANCE 2)
  ;; Compute value at one end of a wire from
  ;; the value at the other end
  IF [AND [WIRE ≡TERMINAL1 ≡OBJECT1 ≡TERMINAL2 ≡OBJECT2]
          [VALUE-OF ≡TERMINAL1 ≡OBJECT1 ≡VALUE]]
  THEN [VALUE-OF  ≡TERMINAL2 ≡OBJECT2 ≡VALUE])

(DEFUN SIMULATE ()
  (TELL [VALUE-OF A P1 3])
  (TELL [VALUE-OF B P1 2])
  ...)
```

```
(DEFRULE DETECT-TERMINAL-CONFLICT (:BACKWARD)
  ;; Infer a conflict from difference of
  ;; observed and simulated values.
  IF [AND [OBSERVED-VALUE-OF ≡TERMINAL ≡OBJECT ≡OBSERVED-VALUE]
          [VALUE-OF ≡TERMINAL ≡OBJECT ≡COMPUTED-VALUE]
          (≠ ≡OBSERVED-VALUE ≡COMPUTED-VALUE)]
  THEN [CONFLICT-AT ≡TERMINAL ≡OBJECT
                    ≡OBSERVED-VALUE ≡COMPUTED-VALUE])

(DEFUN FIND-CANDIDATES ()
  ;; find candidates that explain all the conflicts
  (LET ((SUPPORT-SETS NIL))
    (ASK [CONFLICT-AT ≡TERMINAL ≡OBJECT
                      ≡OBSERVED-VALUE ≡COMPUTED-VALUE]
         #'(LAMBDA (CONFLICT)
             ;; for each conflict derived,
             ;; record its set of supporting assumptions
             (PUSH (SUPPORT CONFLICT :ASSUMPTION) SUPPORT-SETS)))
    ;; now take the intersection of all such sets
    (APPLY #'INTERSECTION SUPPORT-SETS)))

(DEFUN SETUP ()
  (TELL [TYPE-OF M1 MULTIPLIER])
  (TELL [STATUS-OF M1 WORKING] :JUSTIFICATION 'ASSUMPTION)
  ...
  (TELL [WIRE OUTPUT PRODUCT M1 INPUT ADDEND A1]))
```

**Figure 2: Joshua Code for The Trouble Shooter**

Joshua provides well-crafted default implementations for all of its standard facilities. Discrimination networks are used for data and rule indexing. Forward chaining rules use a Rete network [Forgy] to merge the bindings from matching the separate trigger patterns. There is a rule compiler that transforms the rule's patterns and actions into Lisp code. Using these default facilities, we achieve a rule-firing rate of about 120 rules/second while running the trouble-shooting example on a Symbolics 3640. This is comparable with other well-implemented tools.

## 3.1. The Problem

Joshua maintains several internal meters, one of which indicates that during the execution of the trouble-shooting procedure the Rete Network's efficiency was only 5%. This means that the system wasted a lot of effort trying to trigger rules. One reason for this is clear: The WIRE rule contains two trigger patterns, each of which contains only variables. This means that the Rete network will try to merge *every* WIRE statement with *every* VALUE-OF statement, failing in most cases. There are several other mismatches between the problem and the implementation structure. The uniform statement-oriented syntax of Joshua is a reasonable means for *expressing* the problem solving strategy. However, the statement-oriented indexing scheme needed to support this expressive generality provides a poor *implementation* for our specific problem since it cannot exploit its constraints.

A constraint language framework like that in [Davis & Shrobe] which uses data structures mirroring the connectivity and topology of the circuit would better exploit the limitations of our problem domain. However, we want to avoid changing our rules or our trouble-shooting procedures since these constitute the "knowledge level" of the program. Finally, we want to avoid writing a large amount of code simply to take advantage of an existing set of data structures. The key to achieving all three of our goals simultaneously is Joshua's Protocol of Inference.

# 4. The Protocol of Inference

The structure of the Protocol is shown in Figure 3; each step of the protocol corresponds to a generic function that dispatches on the type of the statement being processed. We implement each statement as a *Instance* of a *class*, where the class corresponds to the Predicate of the statement. The classes are organized in an inheritance lattice with each class providing some protocol methods and inheriting others from more abstract classes. (In our current implementation, the classes are flavors and the statements are flavor instances).

For example, the statement [VALUE-OF ADDEND A1 10] is an instance of the VALUE-OF class; this class inherits from the class for PREDICATION (all statements inherit from this class); in the default implementation it also inherits from the DN-MODEL class which provides discrimination-network data indexing. The PREDICATION class provides the gross structure of the ASK and TELL protocol steps in its ASK and TELL methods. The DN-MODEL class provides a specific kind of data indexing by supplying methods for the INSERT and LOCATE-TRIGGER protocol steps which determine where data and rules are stored.

The generic function for each protocol step dispatches on the type of a statement to determine, using the inheritance lattice, which method to run. For example, the generic function for the TELL protocol step when applied to the predication [VALUE-OF ADDEND A1 10] executes the TELL method inherited from the PREDICATION class. This TELL method calls several other generic functions, in particular the one for the INSERT protocol step. This method is inherited from the more specific DN-MODEL class. In the Flavors implementation used by Joshua, inheritance is a compile-time operation which incurs no run-time cost.

The Protocol has major steps for TELL, ASK, the TRUTH-MAINTENANCE entry points, and RULE-COMPILATION; it has minor steps corresponding to the details of how each of the major actions is performed. For example, TELL is concerned with installing new information. Its components are JUSTIFY, which is the interface to the TMS, INSERT, which manages the actual data indexing, and MAP-OVER-FORWARD-TRIGGERS which invokes forward-chaining rules using the Rete network. This, in turn, relies on the LOCATE-TRIGGER protocol step which manages the indexing of rules.

The advantage of exposing this structure is modularity: If one only wants to modify how the data is indexed, one doesn't have to reimplement all the behavior of TELL. Instead one need only provide a new INSERT method; the rest of the behavior can be inherited from the defaults provided with the system. If one wants to modify how rules are indexed, one only has to provide a LOCATE-TRIGGER method. The implementor should define these methods at a place in the lattice of classes so that only the desired statements inherit the new behavior. If, for example, there is a specialized indexing scheme which works well for a restricted class of statements we can easily make that set of statements take advantage of the technique, while all other statements continue to use the more general techniques provided as the system default.

# 5. Using The Protocol of Inference

Figure 4 shows an implementation technique for the trouble-shooting example which is similar to those used for constraint-languages. These structures can be thought of as a set of frames and slots. The frames are used to represent objects, e.g. ADDER-1, and classes of objects, e.g. ADDER. The slots are used to represent terminals, e.g. the ADDEND of ADDER-1; the facets of the slot are used to represent the value of the signal present at the terminal, the set of other terminals wired to it and the set of relevant rules.

- Tell: installs new information.
  - Justify: the interface to the TMS.
  - Insert: manages the actual data indexing.
  - Map-Over-Forward-Triggers: finds and invokes rules.
    - Locate-Trigger: manages the indexing to locate relevant rules.
- Ask: retrieves known or implied data.
  - Fetch: manages the data indexing to find statements which might unify with the query
  - Map-Over-Backward-Triggers: finds and runs relevant rules.
    - Locate-Trigger: manages the indexing to locate relevant rules.
- TMS Protocol: Manages Deductive Dependencies
  - Justify: installs a new TMS justification
    - Notice-Truth-Value-Change: allows special processing when statements change truth value.
  - Retract: removes a justification.
    - Notice-Truth-Value-Change: as above.
  - Explain: prints an explanation of the reason for believing a statement.
  - Support: finds the set of facts or assumptions that a statement depends on.
- Rule Indexing Protocol
  - Add-Forward-Trigger
  - Remove-Forward-Trigger
  - Add-Backward-Trigger
  - Remove-Backward-Trigger
    - Trigger-Location: used by all four of the above.
- Rule Customization Protocol
  - Compile-Forward-Trigger: the hook to provide your own matcher for a forward-chaining rule.
    - Positions-Matcher-Can-Skip: informs the match compiler that the data indexing scheme guarantees that certain positions of the statement already match the pattern, so that less match code can be generated.
  - Compile-Backward-Trigger: same for backward-chaining rules.
    - Positions-Matcher-Can-Skip: as above.
  - Compile-Forward-Action: tailors the behavior of a statement in the THEN part of a forward-chaining rule.
  - Compile-Backward-Action: tailors the behavior of a statement in the IF part of a backward-chaining rule.

**Figure 3: The Protocol of Inference**

This representation exploits the object-oriented nature of the problem in several ways: First, the topology of the data structures is identical to that of the circuit; to find what other terminals are connected to the ADDEND of ADDER-1 one only need fetch the WIRES facet of the terminal. Second, facts are indexed locally. To find the value of the signal at the ADDEND of ADDER-1, one need only find ADDER-1 and then find its ADDEND slot. Third, rules are indexed locally. To find a rule which is triggered by the statement [VALUE-OF ADDEND A1 10], one only need find the A1 frame, follows its AKO link to the class ADDER, and then find ADDER's addend slot. Thus, to add or retrieve information or to draw an inference one need only follow a small number of pointers. In particular, notice that wires are represented by *direct* links between connected terminals, instead of the troublesome WIRE rule shown in Figure 2. These data structures can be implemented easily using a frame-like subsystem provided with Joshua[1].

However, let us imagine that we already have an implementation of a constraint language and then consider what we would need to do to make Joshua able to incorporate it. The trouble-shooting program has two broad categories of statements: The first category consists of TYPE-OF, and WIRE statements which describe the topology of the circuit. The second category includes VALUE-OF and OBSERVED-VALUE-OF statements which carry information about the value (or inferred value) of signals in the circuit. CONFLICT-AT statements also fall in the category, since they capture a discrepancy between the predicted and observed values.

The trouble-shooting program is primarily an antecedent reasoning system, so our attention will be focused on what methods we need to provide for the component steps of the TELL protocol. For the first category of statements our strategy will be as follows: When we TELL a TYPE-OF statement, e.g. [TYPE-OF A1 ADDER], we will build a frame representing A1 that is an instance of the ADDER frame. This frame has slots for each of A1's terminals, and each of these has several facets, one of which is the WIRES facet. When we TELL a WIRE statement, e.g. [WIRE PRODUCT M1 ADDEND A1], we will add pointers to the WIRES facet of both mentioned terminals so that the PRODUCT of M1 points to the ADDEND of A1 and vice versa. The INSERT protocol method is the right level of the TELL protocol to control this. Similarly, we only need to provide an INSERT method for WIRE statements which updates the WIRES facet of the appropriate slot. Also for each of these statement types we provide a FETCH method (the step of the ASK protocol responsible for locating the data) so that we can retrieve the data. Other than this, all processing of these statements uses the provided facilities.

The second category of statements deals with signal values. Our strategy for these is as follows: We will store VALUE-OF and OBSERVED-VALUE-OF statements in the appropriate named TERMINAL of the circuit; to do this we need only provide an INSERT protocol method for VALUE-OF and OBSERVED-VALUE-OF statements. Since the constraint-language provides a means for locating a named terminal, we need only have our protocol method call this procedure.

[1]For space reasons, we won't discuss the Joshua flavor-based frame system here.

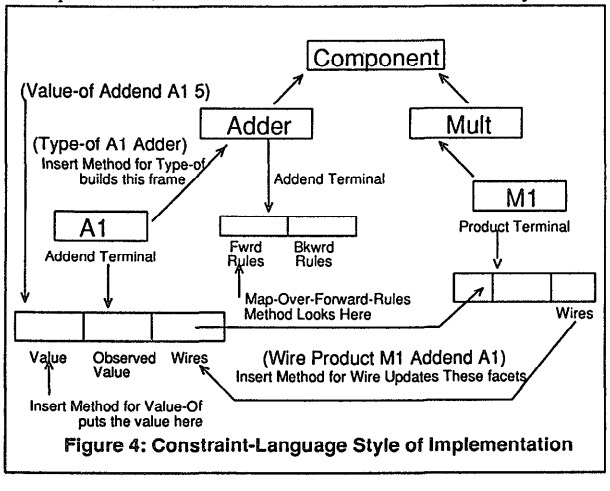


**Figure 4: Constraint-Language Style of Implementation**

In addition, we want to store our rules locally; for example, a rule about adders with the pattern [VALUE-OF ≡ ADDEND ≡ A1 ≡ V], should store its trigger in the FWRD-RULES facet of the ADDEND slot of the ADDER frame. To do this we only need to provide a LOCATE-TRIGGERS protocol method. The LOCATE-TRIGGERS protocol method is used by the protocol steps for installing rules and for fetching them; thus this one modification changes the complete rule data indexing scheme for this type of statement.

## 5.1. Reasoning Within The Model

These data structures can perform certain deductions far more efficiently that can our rules. Since the data structures exactly mirror the topology of the wires in the circuit being modelled we should use them to model the propagatation of signals along wires. To do this, we only need to add a small amount of code to the INSERT method for VALUE-OF and OBSERVED-VALUE-OF statements. This code examines the WIRES facet of the terminal mentioned in the statement and then propagates the information to the connected terminals, by TELLing a new statement describing the value at the connected terminal. For example, suppose we TELL the system that the value of the PRODUCT of M1 is 6, and this terminal is connected to the ADDEND of A1. The INSERT protocol method for VALUE-OF statements, will then TELL the system that the value of the ADDEND of A1 is also 6. (The reason this doesn't create an infinite loop is that part of the contract for INSERT methods is that they must first check to see if the data is already present; if so they must simply return the stored data).

CONFLICT-AT statements are also more efficiently deduced within the model. A CONFLICT-AT statement should be deduced anytime the VALUE and OBSERVED-VALUE at a terminal disagree. To perform this inference, we again add code to the INSERT protocol method for each of these statements; this checks to see if we know both the VALUE and OBSERVED-VALUE at this terminal. If so, and if they disagree, then we TELL the appropriate CONFLICT-AT statement.

## 5.2. An Incorporated Constraint Language

In summary, our system now has the following specialized behavior. When we make an assertion about the TYPE-OF an object, we create a representation of this object and its electrical terminals. This representation is situated in a taxonomic hierarchy below the node representing its type. For example, (TELL [TYPE-OF A1 ADDER]) creates an object of type ADDER and names it A1. When we TELL that a wire connects two terminals, we update facets of their corresponding TERMINALs so that they contain direct pointers to each other. When we make a statement about the value at a terminal (e.g., [VALUE-OF ADDEND A1 2]), we locate the terminal data structure by first locating the object A1 and then finding the terminal named ADDEND. This terminal has direct pointers to all the other terminals that are connected to it, so we just follow these pointers and update the value at these other terminals as well. For each terminal that we update, we find the rules which might trigger by looking in the FWRD-RULES facet.

The whole system behaves like a constraint propagation system. However, it is not just *behaving like* a constraint propagator, it *is* a constraint propagation system embedded in the more general Joshua framework. Incorporating this constraint propagator only required a few small protocol methods, without reprogramming any "knowledge-level" structures such as our rules and trouble-shooting procedures. Finally, although we've tailored this part of our system to the style of reasoning found in constraint language, nothing we've done prevents us from using more general purpose facilities in other parts of our system.

# 6. Proof of the Pudding: The Power of Modeling

We refer to the process we've gone through as *modeling*. Like the logician's notion of modeling, it maps statements to the objects to which they refer. The fact that each of these protocol steps can be tailored for any class of statements has allowed us to easily implement the object-oriented data-indexing and rule retrieval scheme shown in Figure 4. We have presented enough of the details to show

the relative ease with which these facilities can be used. It is worth noting that we did not change the way that Joshua manages *all* assertions, only those which we felt *needed* special handling. Other statements are handled in the default manner. Given the constraint language representation for circuits, we needed to write about six protocol methods, each of them containing only a few lines of code.

The following table, comparing the default and modeled implementations, illustrates the power of this approach:

| Statistic | General | Specialized |
|---|---|---|
| Rules fired | 37 | 5 |
| Time | 0.302 sec | 0.089 sec |
| Rules/Sec | 122.54 | 56.15 |
| Normalized[2] Rules/Sec | 122.54 | 415.51 |
| Merging | 42/774 (5%) | 10/26 (38%) |

Several facts are worth noticing here. First, the number of rule executions went down by a factor of 7. This is because more of the reasoning happens *within the models*, i.e., is performed by the specialized procedures. In particular, there is no longer a need for rules to propagate information along the wires. Second, the program ran over 3 times faster. Finally, the specialized implementation is much more selective; far fewer attempts are made to merge assertions through the Rete network and, of these, a much higher percentage succeed.

### 6.1. Incorporating Other Existing Tools

So far we've seen an example of how the process of providing specialized protocol steps can lead to dramatic improvements in efficiency. But this is not the only advantage. It is probably more significant that modeling provides a simple means for incorporating an existing tool which was designed outside the Joshua context.

One brief example of this is the incorporation of an ATMS. The default TMS in Joshua is similar to that in [McAllester]. However, one of us (Hamscher) had previously implemented an ATMS [deKleer] for use in his research. Sometime later, when he decided to use Joshua as the general framework for his project, he also decided to continue to use his existing ATMS code. Interfacing this code involved implementing about five protocol methods.

## 7. Comparison to other Approaches

The core problem addressed by Joshua has been studied widely in AI. Much of the literature on Meta-Level reasoning, for example [Russel] has been motivated by the need to combine disparate systems into a coherent whole. Compared to Joshua, most of these systems pay a price at run-time for their flexibility since, at least in principle, they must deduce how to do any deduction. The Krypton [Brachman, et al.] system also has the goal of combining disparate facilities, using a theorem-prover as the glue. Theory Resolution [Stickel] provides the theoretical framework for this system. Also [Nelson & Oppen] describes a means for combining disparate decision procedures into a larger, uniform decision procedure. Joshua lacks the theoretical foundations of these systems. However, it seems to provide a broader and more flexible framework which provides stronger guidance for how to actually implement a heterogeneous system.

Our system is quite similar to the *Virtual Collection of Assertions* notion presented in [Kornfeld], but differs in several ways. Joshua provides more complete integration with Lisp as well as a set of high performance techniques available as defaults. In addition, the Protocol of Inference provides a structure and granularity of control not present in Kornfeld's system.

[2]This scales up the firing rate of the modeled version by 37/5, since it does 37 rules' worth of work in just 5 rule firings. The "hidden" rules are done *in the representation.*

## 8. Conclusions

AI has suffered from an inability to consolidate its gains in the form of a programming system which is encompassing and which allows the abstraction level of our problem solving systems to grow. A key failure of previous systems has been their inability to provide strong paradigmatic guidance without implementational handcuffs. Joshua addresses these problems in several ways.

- It **removes syntactic barriers.** Joshua's deductive facilities and Lisp are closely integrated and easily mixed.
- Joshua is organized around a **uniformly accessible heterogeneous database,** whose interface is the two generic functions ASK and TELL. These provide the abstraction level necessary to allow statements to be stored in whatever manner is most convenient and efficient. Special purpose inference procedures can be invoked at this interface.
- Joshua's core routines are carefully structured into a **Protocol of Inference.** This allows a Joshua programmer to use specialized data structures and procedures without having to abandon the general purpose framework. Specialized approaches can be provided by supplying only a few simple methods.
- The Protocol of Inference also facilitates the **assimilation of existing facilities** which enriches the Joshua environment.

Joshua, therefore, creates the possibility of an integrating facility which can combine disparate AI techniques into a coherent total system.

## 9. References


Brachman, R.J., Fikes, R.E., and Levesque, H.J., 1983. "Krypton: A Functional Approach to Knowledge Representation," IEEE Computer, Vol 16. No. 10, October, 1983, pp. 67-73.

Davis, R. and Shrobe, H., 1983. "Representing Structure and Behavior of Digital Hardware". **Computer** vol 16 number 10, October 1983.

Davis, R., Shrobe, H., Hamscher, W., Wieckert, K. Shirley, M. and Polit, S., 1982. "Diagnosis Based on Descriptions of Structure and Function", **AAAI-82** pp. 137-142, Pittsburgh, PA.

deKleer, J. and Williams, B., 1986. "Reasoning About Multiple Faults", **AAAI-86,** pp. 132-139. Philadelphia, Pa.

deKleer, J., 1986. "An Assumption-Based Truth Maintenance System", **Artificial Intelligence** 28:127-162.

Forgy, C. 1982. "RETE: A fast Algorithm for the many pattern/many object pattern match problem." **Artificial Intelligence** 19:17-38.

Genesereth, M.R., 1984. "The Use of Design Descriptions in Automated Diagnosis", **Artificial Intelligence** 24:411-436.

Kornfeld, W.A., 1981. **Concepts in Parallel Problem Solving.** Ph.D. Thesis, MIT Department of Electrical Engineering and Computer Science. October 1981.

McAllester, D.A., 1980. "An Outlook on Truth Maintenance". MIT Artificial Intelligence Laboratory Memo 551. MIT, Cambridge Mass.

Nelson, G. and Oppen, D., 1978. "A Simplifier Based on Efficient Decision Algorithms", Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, Tucson, Arizona, January 1978 pp. 141-150.

Russel, S., 1985. **The Compleat Guide to MRS.** Stanford Knowledge Systems Laboratory Report No. KSL-85-12. Stanford Knowledge Systems Laboratory, Stanford CA.

Stickel, M.E., 1983. "Theory Resolution: Building in Nonequational Theories." **AAAI-83** Washington, D.C. August 1983.

Sussman, G.J. and Steele, G.L. Jr., 1980. "CONSTRAINTS: A Language for expressing almost-hierarchical descriptions" **Artificial Intelligence** 14:1-39.