MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper No. 326 May, 1989

Principles of Knowledge Representation and Reasoning in the **FRAPPE** System

Yishai A. Feldman and Charles Rich

The purpose of this paper is to elucidate the following four important architectural principles of knowledge representation and reasoning with the example of an implemented system: limited reasoning, truth maintenance, hybrid architecture, and many sorted logic.

Introduction

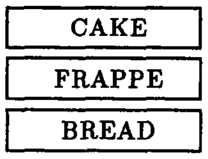
The purpose of this paper is to elucidate the following four important architectural principles of knowledge representation and reasoning with the example of an implemented system, called FRAPPE:

- limited reasoning
- truth maintenance
- hybrid architecture
- many sorted logic

Although each of these principles has been illustrated in other systems, we believe FRAPPE is unique in embodying all of them in a single system.

Before discussing each of these principles in detail, we begin with a brief history of the system. The FRAPPE system has evolved in the context of the Programmer's Apprentice project [17]. One of the goals of the project is to develop an intelligent programming tool to assist in all phases of the programming process. Such a tool requires, among other things, powerful facilities for representing structured artifacts (such as programs, specifications, and requirements) at various levels of abstraction and reasoning about their properties.

The original kernel of FRAPPE was a system for propositional reasoning with equality developed by McAllester [13] in 1982. This was modified and extended by the authors, resulting in a system called BREAD (for "Basic REAsoning Device") completed in 1984. FRAPPE (for "FRAmes in a ProPositional Environment") was built on top of BREAD between 1984 and 1987 to provide an additional range of broadly applicable facilities, such as frames and algebraic reasoning. Finally, we have recently completed building on top of FRAPPE a layer of specialized facilities for representing and reasoning about programming concepts, such as data flow, control flow and side effects. This layer is called CAKE (see below).



There are currently five graduate students using BREAD/FRAPPE/CAKE for their thesis work.

In this paper we discuss only the FRAPPE system (including BREAD), since it is the most broadly applicable part of our work. (CAKE will be described elsewhere.) For each principle noted above, we discuss its motivation, how the principle is embodied in FRAPPE, and what has been learned.

Limited Reasoning

Complete decision procedures for most general reasoning problems, such as propositional logic, require time in the worst case exponential in the size of the reasoning data base. In early AI systems, this difficulty was avoided by limiting the power of reasoners in various ad hoc ways, such as by limiting the machine resources consumed in answering a particular question.

More recently, researchers have been searching for more principled ways to limit reasoning, with two kinds of motivation. Some limited reasoning principles are motivated by architectural concerns: To use a reasoner as a module in building larger systems, we need to have a clear idea of what it will and will not do. For example, work on limited forms of subsumption [2] is mostly motivated by such concerns. Other kinds of limited reasoning, such as recent work on "vivid" representations [10], are motivated primarily by cognitive modelling concerns.

At the heart of FRAPPE is a limited reasoner for propositional logic in conjunctive normal form. Given a set of clauses, it is limited to performing resolutions, such as the following example, in which the resolvent and all but one of the inputs are unit clauses.

$$\begin{array}{l} A \vee \overline{B} \vee C \\ \overline{A} \\ B \\ \hline C \end{array}$$

This limited form of deduction is equivalent to achieving arc consistency in a constraint satisfaction problem [11] in Boolean variables. The algorithm FRAPPE uses for this (due to McAllester [12]) can be thought of as a constraint propagation network [19], in which there is a cell corresponding to each atomic proposition (e.g., A, B, and C above),[1] and the values stored in the cells are Boolean (cells may also have no value). The cells are wired together with constraint boxes corresponding to the non-unit clauses.

The constraint propagation algorithm guarantees that, if the cells corresponding to propositions in unit clauses are initialized to *true* or *false* (depending on the sign of the literal), then values will be assigned to other cells in the network, such that each cell will have the value *true* (or *false*) if and only if the corresponding proposition (or its negation) follows by the kind of unit resolution described above. The algorithm also detects when a constraint box is violated, which in this case corresponds to deriving the empty clause, meaning that a contradiction exists.

The constraint propagation algorithm requires time in the worst case linear in the number of constraints. Moreover, it is an incremental algorithm, which is particularly suitable for interactive applications.

[1]As we will see below, an atomic proposition may also be the application of a Boolean-valued function (predicate) to ground terms, such as $P(n)$ and $Q(m, f(n))$, where m and n are individuals.

This form of limited reasoning is motivated by an architecture in which systems built on top of FRAPPE view it as an "active data base," i.e., one that maintains certain integrity constraints. Specifically, we view FRAPPE as a data base of propositions in which a limited form of logical consistency and completeness is maintained. It is crucial to the active data base metaphor that whatever processing automatically takes place as part of update and retrieval be strictly limited. We don't want any algorithms tightly coupled to the data base that can take an arbitrarily long time.

It is also desirable in general to have the option of increasing the power (and of course cost) of a limited reasoner in explicitly controlled steps, approaching a complete decision procedure, if one exists. FRAPPE can be requested to "try harder" by attempting a refutation proof for a specified proposition: The value of the corresponding cell is temporarily set to *true* (or *false*). If the propagation of this value leads to a contradiction (violated constraint), then the opposite value is deduced. This can be made a complete (and exponential) decision procedure for propositional logic by trying all combinations of values for all cells with no value.

The central outstanding question regarding limited reasoners in general is whether defensible limiting principles can be found. In the case of cognitive modelling, a principle can be defended by human experiments. In the architectural approach, the question is whether system builders can understand a principle well enough to build a system with reliable behavior. The final verdict on the limited reasoning principle in FRAPPE is not yet in: Some users have found it obscure; others have found it quite workable.

Truth Maintenance

Although truth maintenance techniques [7] have been used for specific kinds of problem solving for a long time, they have only recently been included as an integral part of general-purpose knowledge representation and reasoning systems (see, for example, [1, 18]). The motivation for this development is the realization that being able to explain, justify, and change beliefs, and to manage contradictory beliefs and multiple contexts of belief is fundamental to intelligent systems in general.

FRAPPE's Boolean constraint propagation network (described in the preceding section) includes a truth maintenance system which records justifications for all derived values. The unit clauses in the reasoning data base are treated as premises, which can be asserted or retracted by changing or removing the value of the corresponding cell.[2] The non-unit clauses (corresponding to constraint boxes in the network) cannot be changed.

To obtain the effect of retractable non-unit clauses, we introduce function symbols for Boolean connectives with the appropriate axioms. For example, the retractable

[2]This is a slight simplification: One may also create a constraint box for a unit clause, making the corresponding literal non-retractable.

clause $A \vee B$ is simulated by creating the atomic proposition or(A, B) and adding the following clauses to the data base.

$$\overline{\text{or}(A,B)} \vee A \vee B$$
$$\overline{A} \vee \text{or}(A,B)$$
$$\overline{B} \vee \text{or}(A,B)$$

All of the specialized reasoners built into FRAPPE (see next section) interface with each other by adding clauses to the propositional data base. This results in a uniform truth maintenance principle throughout the system. Facilities are also provided so that further additions to FRAPPE can obey this discipline.

One lesson we have learned, however, from implementing a number of specialized reasoners using the truth maintenance principle is that there can be huge performance gains from providing the option to declare certain facts non-retractable. Note that we do not mean allowing a reasoner to install a fact without the correct justification—we mean declaring that a given premise will not be retracted, which is something the system can monitor and enforce.

A good example of this phenomenon is commutativity:

$$\text{commutative}(f) \Rightarrow f(x,y) = f(y,x).$$

In the general case, if the premise commutative(f) is potentially retractable, then one has to instantiate this axiom for each matching pair of applications of f. However, if the commutativity of f is guaranteed not to change (for example, you may never want to change the meaning of the symbol $+$), then this axiom can be implemented by canonicalizing the order of arguments to applications of f at the time they are created. This eliminates a large overhead in extra terms and clauses.

The main outstanding architectural question regarding truth maintenance is when and how to "garbage collect" justifications. At the moment, FRAPPE stores every justification created for as long as the system runs. It is clear that this approach will not scale up to large applications.

Other outstanding questions regarding truth maintenance are concerned more with the use of justifications in particular kinds of problem solving, such as how to search large context spaces efficiently [6].

Hybrid Architecture

A hybrid knowledge representation and reasoning system is one in which two or more fundamentally different algorithms and data abstractions are used. In this sense, it is fair to say that all practical systems are in fact hybrid. (For example, no practical system tries to deduce that $2+2=4$ from the axioms of arithmetic.) The usual motivation for a hybrid architecture is to take advantage of specialized methods for many common kinds of reasoning that are much more efficient than a single universal method can ever be.

The main issue in this area is, therefore, (as with limited reasoners) to develop a principled as opposed to ad hoc approach. We will return to this issue after briefly describing the specialized reasoners in FRAPPE (see also [3]). In FRAPPE, we have implemented ground reasoners[3] for the following theories:

- equality
- simple algebraic properties
- frames
- sets

The ground reasoner for equality in FRAPPE is complete. Congruence closure is a well studied problem (see, for example, [16]), for which quite efficient ($n \log n$) algorithms have been developed. The algorithm used in FRAPPE is somewhat less than optimal due to some tradeoffs that have been made to facilitate incremental assertion and retraction of equalities, and the recording of justifications.

Any operator symbol in FRAPPE can be asserted (perhaps non-retractably) to have one or more of the following simple algebraic properties: transitive, associative, commutative, reflexive, symmetric, antisymmetric, involutive, idempotent. Efficient (worst case linear or $n \log n$), complete ground reasoners have been implemented for each of these theories using a combination of special-purpose data structures and general-purpose system-building facilities in FRAPPE, such as pattern-directed invocation and other kinds of demons.

A frame instance in FRAPPE is essentially an n-tuple with named rather than numbered components (slots). Slot value restrictions, constraints between slots, and inheritance between frame classes are achieved via FRAPPE's sort structure described in the next section. FRAPPE's ground reasoning is complete with respect to constructors, selectors, and equality between frame instances.

Finally, the largest investment we have made in a specialized reasoner for FRAPPE has been for ground reasoning with sets, i.e., the language of $\in$, $\subseteq$, $\cap$, $\cup$, and $'$. This reasoner is not complete (we do not yet have a simple characterization of its limited reasoning—basically it is incomplete with respect to complements, unless you ask it to "try harder").

[3]By a ground reasoner for theory T, we mean a procedure for determining whether a given ground literal in (the language of) T follows from a given set of ground literals in T according to the axioms of T. Note that a complete ground reasoner for T is *not* the same as a complete decision procedure for the quantifier-free theory of T, since the quantifier-free theory of T includes propositional logic as a sub-theory.

The major lesson we learned from the hybrid architecture of FRAPPE is the difficulty of debugging all of the interactions between reasoners on a case-by-case basis. For example, for each pair of complete reasoners, say transitivity and equality, we needed to make sure that the resulting combination is a complete reasoner for the union theory. This is complicated by the fact that the underlying facts may come and go in any order via the truth maintenance system. In the case of combining a complete reasoner with a limited reasoner, say equality and propositional logic, we have the difficulty of giving a principled description of the combined limited reasoner. We also had to worry about the interaction between equality and the basic pattern-directed invocation mechanisms [8]. (A subsidiary moral here is that equality interacts with everything!)

A strong solution to this problem would be a hybrid architecture in which there was a simple interface specification, which if satisfied, would guarantee completeness of the system when a new reasoner is added. Examples of such architectures exist in restricted settings, such as combining decision procedures for disjoint theories (theories which share no non-logical symbols) [15], and the addition of sort reasoners to unification-based systems [9]. Neither of these frameworks, however, deals with the problem arising in FRAPPE of combining complete and limited reasoners, nor includes justifications and changing beliefs as part of the interface.

Barring a general hybrid architecture with strong completeness guarantees, the only other promising approach for a system like FRAPPE is a framework that provides a standard protocol for inference within each reasoner [18] and for communication between reasoners [14]. For example, we imagine that if we were starting to build FRAPPE today, it would be much easier to do using the inference protocol described in [18].

Many Sorted Logic

There are two motivations for using a many sorted logic as the semantic foundation of a knowledge representation and reasoning system. The first is efficiency: Many sorted axiomatizations and proofs using them are typically smaller than the corresponding unsorted axiomatizations and proofs. Furthermore, in a system built using a many sorted logic, much of the taxonomic reasoning in a given problem can be achieved by specialized reasoning mechanisms operating on the sort structure, rather than by general purpose deduction. In this sense, use of a many sorted logic is an example of hybrid reasoning.

A second motivation for using a many sorted logic is more incidental: It has generally been the experience of system builders (confirmed by our work with FRAPPE) that including taxonomic information in the syntax of the logic helps to catch errors and generally manage complexity in the formalization task.

In FRAPPE we have implemented an expressive form of many sorted logic, similar to [4]. Its main features (each discussed briefly below) are:

- a complete Boolean lattice of sorts
- polymorphic sort specifications (overloading)
- sort predicates
- overlapping (sorts in expressions)

A complete Boolean lattice of sorts allows us to express not only subsort relationships, but also disjointness and covering. For example, we can specify that Man and Woman are sorts whose intersection is empty and whose union is Human.

With polymorphic sort specifications, an operator (function or predicate symbol) is given a *set* of functionalities. This allows quite sophisticated inferences to be performed entirely within the sort reasoner. For example, assuming Even and Odd are subsorts of Number, the sort specification for the function + might include, among others, the following functionalities:[4]

$$\begin{array}{l}\text{Number} \times \text{Number} \rightarrow \text{Number}\\ \text{Odd} \times \text{Odd} \rightarrow \text{Even}\\ \text{Even} \times \text{Odd} \rightarrow \text{Odd}\end{array}$$

Using this information, FRAPPE can deduce that the term $((a{+}b){+}((c{+}d){+}e))$ is Odd given that a, b, c, d, and e are all terms of sort Odd.

Note that the use of sort specifications for operators also introduces partial functions into the logic: Assuming there are no other more general functionalities given for +, the above also specifies + to be undefined for non-number arguments. Partial functions are useful in many applications, for example, for representing potentially non-terminating computations.

Introducing sort predicates into the logic allows us to derive sort information from non-sort information and vice versa using general-purpose deduction. For example, a function f might have the following property:

$$\text{Odd}(f(a)) \Leftrightarrow a \neq 0.$$

The availability of sort predicates in FRAPPE is also an example of the principle, discussed earlier in connection with truth maintenance, of providing both a retractable and a non-retractable form for certain kinds of information. Assigning the sort Odd to a term allows this information to be exploited by the sort reasoner, but is not retractable. On the other hand, propositions such as $\text{Odd}(f(a))$ are retractable, but are more expensive to reason with. (Note that assigning a sort to a term does make the corresponding proposition true.)

[4]This example is taken from [5].

Finally, in assigning a sort to expressions, we compute a result sort as long as the sort of each argument *overlaps* (i.e., has a non-empty intersection) with the desired sort of the argument position. In more restrictive logics, the sort of the argument is required to be a subsort of the argument position. Several examples of the utility of this feature are given in [4] and [5], but are too long to include here.

In summary, our experience with FRAPPE has confirmed the benefits of many sorted logic. We can see no reason ever to use an unsorted logic as the foundation for a knowledge representation and reasoning system.

Conclusion

As well as being a practical platform for our research in automated programming, FRAPPE has also served as an experimental testbed for principles of knowledge representation and reasoning. We expect the system to continue to evolve in both of these dimensions.

FRAPPE is implemented in Common Lisp and is available for experimental use by other research groups by contacting the authors.

References

[1] J. F. Allen and B. W. Bradford. The rhetorical knowledge representation system: A user's manual (for Rhet version 14.0). Technical Report 238, U. of Rochester, Dept. of Comp. Sci., February 1988.

[2] R. J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *Proc. 4th National Conf. on Artificial Intelligence*, Austin, TX, August 1984.

[3] D. Brotsky and C. Rich. Issues in the design of hybrid knowledge representation and reasoning systems. In *Proc. of the Workshop on Theoretical Issues in Natural Language Understanding*, Halifax, Nova Scotia, May 1985.

[4] A. G. Cohn. A more expressive formulation of many sorted logic. *Journal of Automated Reasoning*, 3(2):113–200, June 1987.

[5] A. G. Cohn. Many many sorted logics. In *AAAI Workshop on Principles of Hybrid Reasoning*, St. Paul, MN, August 1988.

[6] J. De Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28(2):197–224, March 1986.

[7] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[8] Y. A. Feldman and C. Rich. Pattern-directed invocation with changing equalities. Memo 1017, MIT Artificial Intelligence Lab., May 1988.

[9] A. M. Frisch. A general framework for sorted deduction: Fundamental results on hybrid reasoning. In *Proc. 1st Int. Conf. on Principles of Knowledge Representation and Reasoning*, Toronto, Canada, May 1989. Submitted.

[10] H. J. Levesque. Making believers out of computers. *Artificial Intelligence*, 30(1):81–108, October 1986.

[11] A. K. Macworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, February 1977.

[12] D. A. McAllester. An outlook on truth maintenance. Memo 551, MIT Artificial Intelligence Lab., August 1980.

[13] D. A. McAllester. Reasoning utility package user's manual. Memo 667, MIT Artificial Intelligence Lab., April 1982.

[14] S. A. Miller and L. K. Schubert. Using specialists to accelerate general reasoning. In *Proc. 7th National Conf. on Artificial Intelligence*, St. Paul, MN, August 1988.

[15] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2), October 1979.

[16] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, April 1980.

[17] C. Rich and R. C. Waters. The Programmer's Apprentice: A research overview. *IEEE Computer*, 21(11), November 1988. Also published as MIT AI Memo 1004.

[18] S. Rowley, H. E. Shrobe, and R. Cassels. Joshua: Uniform access to heterogenous knowledge structures or why joshing is better than conniving or planning. In *Proc. 6th National Conf. on Artificial Intelligence*, Seattle, WN, August 1987.

[19] G. J. Sussman and G. L. Steele. CONSTRAINTS—A language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–40, 1980.