Towards a General Framework for Composing Disjunctive and Iterative Macro-operators

Peter Shell*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Jaime Carbonell
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Inducing disjunctive and iterative macro-operators from empirical problem-solving traces provides a more powerful knowledge compilation method than simple linear macro-operators. Whereas earlier work focused on *when* to create iterative macro-operators, this paper addresses *how* to form them, combining proven optimization methods such as extraction of loop invariants, with techniques for further optimizing RETE-match efficiency. The disjunctive and iterative composition processes have been implemented in FERMI and its underlying production system language. Empirical results confirm substantial rule-match speedups and system performance improvements in different application domains.

1 Introduction: The Nature of Disjunctive and Iterative Macro-operators

Whereas classical AI techniques for problem solving and planning require vast amounts of search to produce viable solutions for even moderately complex problems, humans typically require much less search as they accrue experience over time in any given domain. Inspired by this ubiquitous observation, researchers in various subdisciplines of AI have sought methods of encapsulating more knowledge to reduce search. These methods range from expert systems, where all knowledge is laboriously handcoded at the outset, to machine learning approaches, where incrementally accumulated experience is stored, processed and compiled into a generalized reusable form.

Machine learning approaches to knowledge compilation include analogical and case-based reasoning [Carbonell, 1983, Carbonell, 1986, Schank, 1982, Schank, 1983, Doyle, 1984, Hammond, 1987] where past experience is recycled directly, as well as the compilation of new rules distilling essential aspects of past experience. The latter methods range from explanation-based generalization [DeJong & Mooney, 1986, Mitchell *et al*, 1986, Shavlik & DeJong, 1987, Minton & Carbonell, 1987, Minton *et al*, 1988], to generalized chunking [Newell, 1981, Laird *et al*, 1986], and all forms of macro-operator formation [Fikes, 1971, Anderson, 1983, Korf, 1985, Minton, 1985, Cheng & Carbonell, 1986]. In all cases, the objective is to transform knowledge into a much more efficient, operational form, using past experience and domain analysis as primary resources.

Although the basic idea of a macro-operator has been well-defined since the days of STRIPS [Fikes, 1971], their full potential is only recently being realized. Until the more recent work on the FERMI problem solver [Cheng & Carbonell, 1986], macro-operators were considered to be nothing more than frozen sequences of atomic operators with several generality and efficiency modifications (e.g.: the triangle-table encoding in STRIPS that enabled all subsequences of larger macro-ops to be encoded compactly and used by the problem solver). Systematic methods for determining when and how to build useful macro-operators were developed after STRIPS, such as exploiting problem decomposability [Korf, 1985] or providing heuristic utility metrics for deciding whether a new macro-op was likely to increase overall problem-solver performance (e.g., Iba's peak-to-peak heuristic [Iba, 1988] and Minton's S-macros and T-macros [Minton, 1985]). Figure 1 depicts the basic process of linear macro-operator formation.

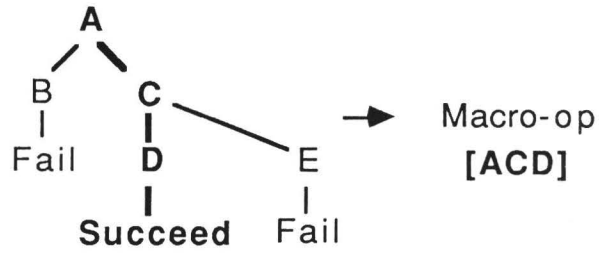

Figure 1: Linear macro-operator formed by concatenating atomic operators found in the solution path to a problem

1.1 Disjunctive Macro-operators

One phenomenon that reduces the effectiveness of simple linear macro-operators is the variation inherent in non-trivial task domains. If multiple variants of a problem present themselves, each may require separate macro-operators that differ on only one or two component steps. Three solutions present themselves: creating a very general macro-operator (which will likely be overly-general), enabling analogical transformation of macro-operators (which incurs high partial-match costs), and producing a disjunctive macro-operator that shares the common parts and diverges only at the key differences. We have chosen the third alternative (see figure 2), generalizing the macro-operator composition algorithm to create disjunctive macro-operators and introducing disjuncts into our rule-description language. Section 2 describes and evaluates the disjunctive composition algorithm in detail.

*This research was supported in part by DARPA contract number F33615-87-C-1499 Amendment 20, and AIP-ONR contract number N00014-86-K-0678-N123.

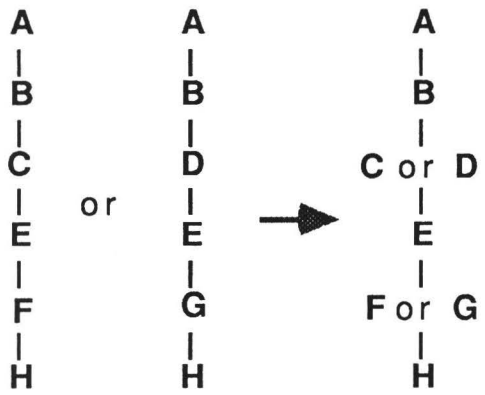

Figure 2: Juxtaposed operator sequences and single disjunctive sequence, solving variants of a problem

1.2 Iterative Macro-operators

Simple linear macro-operators can never generalize to an arbitrary number of applications of atomic operators; for example, different macro-ops must be learned to solve the 2-disk tower of Hanoi, the 3-disk one, etc. Furthermore, both linear macro-ops and Soar chunks [Laird *et al*, 1986] which encode multiple applications of a rule turn polynomial-time matches into exponential ones [Tambe, 1988]. Iterative macro-operators, as depicted in figure 3, prevent these serious problems in FERMI and would solve the SOAR "expensive chunk" problem.

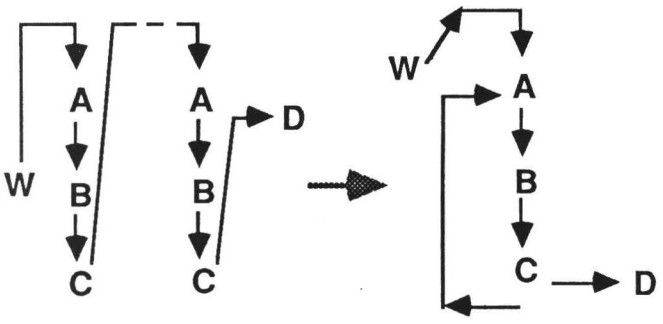

Figure 3: Formation of iterative macro-operators

Previous work on formulating iterative macro-operators focused on detecting situations where such macro-operators would be useful. For instance, Cheng and Carbonell [Cheng & Carbonell, 1986] formulated a method of searching for repeated subgoal-change differences in the solution trace of a complex problem. Riddle [Riddle, 1988], on the other hand, repeatedly decomposed the problem-solution graph by fractioning at the minimal interaction points among the subgoals and operators in the graph, until the remaining fragments were either atomic (single operator) or highly interconnected sets of operators. From each highly-interconnected set, she composed an iterative macro-operator by simply grouping the operators with an exit condition.

In principle, macro-operators (iterative or otherwise) can reduce search in two different ways:

1. A macro-op prevents the application of incorrect or irrelevant operators within the body of the macro-op. If the macro-op were not present, the problem solver could traverse unproductive search paths before backtracking to the correct solution path. However, the reduced state-space search is accomplished at the cost of increasing the match-time search of the larger set of operators (original ones plus the new macro-ops). Utility metrics attempt to optimize this tradeoff [Minton, 1985, Minton *et al*, 1988, Iba, 1988], as does the creation of disjunctive and iterative macro-operators. Since both of the latter are more general than simple linear macro-ops, fewer need be generated.

2. A macro-operator can also compile away temporary computation in the matching process, especially in iterative subsequences. Moreover, when the iterative macro-op is functionally equivalent to multiple invocations of the original operator, the original operator need not be matched. Thus match-level optimizations can reduce "fine-grain search" just like operator-level aggregations can reduce "coarse-grain search".

Although Cheng's and Riddle's methods address the course-granularity search reduction problem, they do not address the equally important issue of optimizing the macro-op itself -- both methods keep the complete primitive operators as inviolate atomic units. If one knows that a set of operators will be applied in a fixed iterative sequence, then transformations to the operators themselves can yield significant efficiency improvements, as we show in section 3. This paper addresses those transformations in detail, and demonstrates speedups greater than six-fold (see section 3.2). In order to synthesize and exploit iterative macro-ops effectively one needs both: a reliable method for recognizing when they should be formulated, and an effective method for composing them efficiently.

2 Composing Disjunctive Operators

Previous work on learning disjunctive concepts has focused on learning in predicate logic representations [Quinlan, 1986, Iba, 1979], rather than learning disjunctive production rules. The BAGGER system [Shavlik & DeJong, 1987] learns a limited form of single-operator iterative rules, allowing only one source rule to be focused on. Also, disjunctions are limited to different instantiations of the same single operator. Cheng [Cheng & Carbonell, 1986] used only linear composition techniques, so when alternate rules were encountered they were not composed together but were placed into an "agenda" and tested sequentially. In the present work, rule sequences which differ by which rules fire as well as how they are instantiated, are composed into a single more efficient disjunctive macro-operator.

Unlike linear macro-operators, disjunctive macro-operators retain the generality of the original rules by encoding all possible paths through these rules. For example, even though the training example depicted in figure 2 demonstrates two solution paths, the single induced disjunctive macro allows all four paths to potentially be taken. Encoding these paths with linear macro-operators requires a rule for each path, and since [Cheng & Carbonell, 1986] doesn't compose disjunctive rules, it would only compose the **A** and **B** operators together. Once all implicit sequences have been learned with a disjunctive operator, the atomic source rules no longer need to be matched, resulting in more savings.

The three phases of forming disjunctive macro-operators are:

1. *Identifying Similar Operator Subsequences.* Different operator sequences that solve the same goal and that share relevant substructures must be noted. These alternate sequences are the basis of the ensuing composition processes.

2. *Identifying Alternate Rules.* The operator sequences identified in the first step are juxtaposed to find shared subsequences and *alternate rules* (see figure 2).

3. *Composing Disjunctive Operators.* The shared and alternate rules are composed into a single disjunctive macro-operator. This allows the elimination of computation of temporary results by disjunctive elements, as well as computation which is redundant across alternate disjunct operators.

Identification of similar operator subsequences was described in [Cheng & Carbonell, 1986]. We describe in this section how we identify the alternate rules, show how we have extended linear operator-composition methods to compose those alternate rules into a single disjunctive macro-operator, and show the usefulness of disjunctive macro-operators with timings.

2.1 Identifying Alternate Rules

Cheng identified alternate rules by the goal they solved. But since our disjunctive composition algorithm acts on rule sequences, we no longer have to assume that every rule matches a goal and instead find these alternate rules by applying sequence-comparing techniques similar to [Sankoff, 1983] to the alternate linear rule traces. For example, in the FERMI algebra module, eliminating an unknown algebraic variable from a system of linear simultaneous equations may be achieved by either of the two sequences of rule applications:

SEQUENCE 1	*Rule #*	SEQUENCE 2
Solve-for-unknown-n	*1*	**Solve-for-unknown-n**
Reduce-eqns&unkns	*2*	**Reduce-eqns&unkns**
Select-var	*3*	**Select-var**
	4	**Rearrange-eqn**
Select-eqn	*5*	**Select-eqn**
Substitute	*6*	**Substitute**
Eliminate	*7*	**Eliminate**

The composition program identifies **Rearrange-eqn** as an *alternate rule* in sequence-2, and pairs the remaining rules. The preconditions and actions in **Rearrange-eqn** at this position are called *Condition Element disjuncts* and *action disjuncts*, respectively.[1]

2.2 Composing Disjunctive Macro-operators

Cheng in [Cheng & Carbonell, 1986] assigned the alternate rule disjuncts to agendas instead of composing them into a disjunctive macro-op. By adding disjunctions to our RETE-based [Forgy, 1982] production-system language FRuleKit [Shell & Carbonell, 1986] and introducing domain-independent disjunctive operator composition, we are able to encode multiple alternate rules in a single disjunctive macro-op. First we will summarize the linear rule-composition algorithm, then show how it is extended to composing disjunctive rules.

2.2.1 Summary of Composition Basics

In the linear composition algorithm on which all previous composing programs are based (see [Anderson, 1983]), one of the following actions are taken with each Condition Element (CE) in the input, or *source* rules:

- *Discard.* CE's that match a WME[2] that was created previously in the rule trace are declared *redundant* and are discarded.

- *Merge.* CE's are *merged* into others (that is, the union of their tests and bindings are taken) if they match a WME that also matched a previous CE in the sequence.

- *Insert.* All other CE's, including absence tests,[3] are retained and *inserted* into the macro-op.

Actions are treated similarly.

2.2.2 Discarding Redundant CE's and Actions

When generalizing composition to multiple rule sequences, note that the same CE is matched by different WME's in each such sequence. Therefore a CE is discarded only if, in every alternate rule sequence, the WME that matched it was created previously in that sequence. If the CE is redundant within some sequences but not others, then the sequences are not considered analogous and a disjunctive macro-op will not be formed. Instead, two linear macro-ops will be created. Similarly, action elements are only labeled redundant if they create WME's that are removed later in all sequences, or if they remove WME's that were created earlier in all sequences.

Another extension to handling redundant CE's concerns production variables bound in an eliminated CE. If these variables are used later in the rule, then an alternate expression for them must be found. If the eliminated variable can not be re-expressed in terms of an inserted non-disjunct action or condition element, then it must be found in inserted action or CE disjuncts. The variable will take on the value of that disjunct expression only if the associated CE disjunct was true.

For example, when composing the **Substitute** rule (see figure 4) into the macro-op, its action references the variable *=F*[4]. This variable is bound to the *:formula* slot of the selected equation that matched this rule. However, the CE that binds *=F* (the second CE) is discarded because the equation that it matches was modified by the previous rule in the sequence (**Select-eqn**). The value of the *:formula* slot depends on whether **Rearrange-eqn** (see figure 5) applies. If **Rearrange-eqn** applies, then it sets the *:formula* slot to *(rearrange =F =E =V)*; otherwise the previous value of this slot (bound to =F1 in the disjunctive rule) is used. Thus the expression for this slot is:

```
If disjunct 1 was true, then: =F1
  else: (rearrange =F2 =E2 =V)
```

Figure 6 shows the entire disjunctive macro-op. Each

[1]"Condition Elements" refer to tests for the existence of objects meeting given constraints.

[2]"WME" is an abbreviation for Working Memory Element, which is the object for which the CE's test.

[3]"Absence tests" test for the absence of any objects in memory matching a pattern.

[4]In FRulekit, production variables are preceded by an '=' sign

pseudo-english line corresponds to one FRuleKit Condition Element or action.

```
RULE Substitute
 CONDITIONS:
  The goal is to form a new equation
    by substitution,
  The selected equation is =E1,
    which has a formula =F,
  The selected variable is =V,
  and =V occurs in a second equation =E2
 ACTIONS:
  Substitute occurrences of =V
    in equation =E2 with =F
```

Figure 4: Atomic rule **Substitute**

```
RULE Rearrange-eqn
 CONDITIONS:
  The goal is to have an equation whose
    LHS is the selected variable,
  The selected variable is =V,
  There is no equation with =V on its LHS,
  But there is an equation =E containing =V
 ACTIONS:
  Put =V on the left-hand-side of =E, and
    set the formula slot of =E
    to (rearrange =F =E =V)
```

Figure 5: Alternate atomic rule **Rearrange-eqn**

2.2.3 Merging CE's and Actions

In essence, when a CE from an alternate rule is merged with another CE, the merged CE is placed into an *<OR>* statement (the <OR> matching statement consists of a list of simple CE's and is considered satisfied if any of its component CE's is satisfied). An analogous procedure is used for actions.

2.2.4 Inserting CE's and Actions

If a CE from an alternate rule is not discarded or merged, then it appears in an <OR> statement in the macro. All corresponding CE's are placed into the same <OR> statement, and their corresponding variables are unified with one another. If an action from an alternate rule is retained, then it and its corresponding actions from corresponding alternate rules are transformed into conditional actions. Again, if one action or test is retained, then all corresponding actions and tests must be retainable; otherwise, the sequences are not considered analogous.

2.2.5 An Annotated Example

The first three rules in the algebra sequence of section 2.1 are composed using non-disjunctive composition techniques, yielding the non-disjunctive CE's in the macro-op. At this point the sequences differ. In sequence-2, the next rule that fired is **Rearrange-eqn** (see figure 5), but no corresponding rule in sequence-1 applied.

The first two preconditions from **Rearrange-eqn** are discarded because the WME's that they matched were created on the previous cycle (by **Select-var**). The next CE is retained because it is an absence test. The *equation* CE of **Rearrange-eqn** matches a WME that a retained *equation* CE already matched, so must be merged. Since this equation CE is from an alternate rule, the retained *equation* CE is taken out of the macro-op and merged with this equation CE.

An <OR> statement whose first disjunct is a list containing the retained absence test and the merged equation CE, is created. Since no rule from sequence-1 corresponds to **Rearrange-eqn**, the second disjunct of the <OR> is just a copy of the simple *equation* CE. After CE's from rules later in the sequence are merged into these disjuncts, the final macro-rule is as shown in figure 6. The number of the source rule from which each CE and action are derived is shown in the first column.

```
RULE Disjunctive-m-solve
 CONDITIONS:
1  The goal is to solve for the unknown,
3  The desired unknown is =U,
1  There are > 1 variables and equations,
3  There is a variable =V that is not =U,
3  There is an equation =E1 containing =V,
  and EITHER:
5    There is another equation =E2 with =V
6      on its LHS and formula =F1 on its RHS
  OR:
4    There is no equation with =V on its LHS,
4    But another equation =E2 contains =V
       and has formula =F2 on its RHS,
7  And =V is in no equation except =E1 or =E2
 ACTIONS:
  If the FIRST disjunct was true,
    then set =FORMULA to: =F1
4    else set it to: (rearrange =F2 =E2 =V)
6  Substitute =V with =FORMULA in =E1, and
6    set the variable list of =E1 to the
     union of the variables in =E1 and =E2
7  Remove =V and =E2
7  Decrement the equation variable counts.
```

Figure 6: Composite disjunctive rule generated by FERMI for eliminating one equation and one unknown

2.3 Efficiency of Disjunctive Operators

We have extended the FRuleKit RETE algorithm so that even tests below CE's in an <OR> statement are shared (for example, the last absence test in figure 6), and to exploit the property that if one disjunct condition is met, then the other disjuncts need not also be matched (details are explained in a forthcoming *CMU Technical Report*). These properties make disjunctive macro-ops significantly faster than their non-disjunctive equivalents. We compare the time spent solving a system of 32 equations by the following rule sets: Cheng's agenda method, where only the first three rules were composed, two linear macro-ops (one incorporating **Rearrange-eqn** and one not), and the single disjunctive macro-op shown in figure 6.

Rule Set	*Time*
Agenda	*22.36 sec.*
Two linear macro-ops	*21.49 sec.*
Disjunctive macro-op	*12.47 sec.*

The two linear macro-ops don't improve on simply composing the first three operators because the same number of tests are shared in both methods. However, the single disjunctive macro-op is almost twice as fast as the old approaches. We will see that iterative operators provide an even greater speedup.

3 Composing Iterative Operators

Generating iterative-operators involves:

1. *Detecting the Repeated Operator.* When a linear rule or disjunctive macro-operator, as described in the previous section, repeats more than once, it is eligible for transformation into an iterative operator.

2. *Encapsulation of the Invariant Structure.* Variables

matched in the preconditions that bind identically each cycle need not be rematched each iteration (similar to detection and extraction of loop invariants in optimizing compilers [Wulf, 1975]). The body of the loop is constructed by reparameterizing the invariant structure (a sequence of one or more atomic operator bodies) with these variables. Furthermore, temporary changes to the RETE matcher can be bypassed, saving more computation.

3. *Solution of Recurrence Relations.* Local data updates between each cycle (such as decrementing an array index, or systematically depleting elements from an unfulfilled goal state) are identified and extracted into a single step external to the loop. More complex or open-form updates remain inside the iteration loop.

4. *Composition of the Total Iterative Macro-operator.* The preconditions, iterative body and exit conditions are deduced from the source atomic operator and compiled into a production.

We have implemented steps 1, 2 and 4 in the FERMI project. This section first describes how we compose the iterative macro-operator with the optimizations of step 2. Then we show how two rule systems were speeded up by iterative operators.

By applying new domain-independent rule-composition and generalization-to-N techniques to successive instances of the source rule applications, an iterative rule is both as general as and more efficient than the source rule. Since the iterative rule is general enough to apply for an arbitrary number of cycles, there is no need to write macro-operators or chunks for specific numbers of repetitions, as in other rule-learning systems ([Fikes, 1971, Anderson, 1983, Laird *et al*, 1986]).

3.1 The Iterative Composition Algorithm

An iterative rule consists of three FRuleKit productions - the *pre-operator*, the *body*, and the *post-operator*. The *pre-operator* has the same left-hand-side as the source rule, except that it also tests for the absence of an "iteration-in-progress" flag. This ensures that the iterative rule will match under the same conditions as the source rule, and that the actions in the *body* production won't cause unnecessary re-matching of the *pre-operator*.

The *body* production performs the iteration. It is equivalent to the source rule except that certain CE's are removed and certain actions affecting the rule matcher are replaced by corresponding actions on LISP variables. Since the *body* stops matching when the source rule would have stopped matching, its exit condition is implicitly the same as the source rule's exit condition. The *post-operator* fires when the *body* no longer can fire (since it matches only on the iterate flag and conflict resolution prefers the more specific *body* production), and transfers the changes from the LISP variables to Working Memory Elements.

Our program moves a much broader class of CE's and actions out of the iterative loop than does [Cheng & Carbonell, 1986]. Cheng's program moved only condition elements that tested constants or variables undergoing simple algebraic transformations, and their corresponding action elements. Our system first classifies CE's as one of:

1. *Precondition-CE.* A CE is a *precondition-CE* if it tests the same WME every cycle of the rule trace. For example, the *current-goal* test in figure 6 is of this type because it tests the same *current-goal* WME each cycle.

2. *Temporary-CE.* If, on each cycle of the rule trace (except the first), the CE matched a WME that was modified on the previous cycle, then it is a *temporary-CE*. For example, the first *equation* test in the disjunctive algebra macro-op (figure 6) tests an *equation* WME, which is modified each cycle by the right-hand-side of that macro-op.

3. *Retained-CE.* Otherwise, the CE is a *retained-CE*. These are CE's that test against new data every cycle, and absence tests. Figure 8 illustrates the CE's that were retained for the algebra example.

Temporary-CE's, their corresponding right-hand-side actions, and *Precondition-CE*'s are not included in the *body*. Since the *precondition-CE* matches the same WME every cycle, we needn't re-match that WME during the loop. The *Temporary-CE* matches WME's which change each cycle, but the changes are made only by the right-hand-side of the *body*, so can be determined by analyzing the right-hand-side actions. These changes are stored in LISP variables instead of WME's (there is one such variable for each slot modified - for example, *Param-U* and *Param-E1* in figure 8), and are restored to the WME's in the *post-operator*. Since the change is only made to the WME once, instead of each cycle as before, significant saving is achieved. The remaining right-hand-side actions are side-effects.

The following additional rules are used for transforming disjunctive operators into iterative operators:

- If a CE inside an <OR> statement is retained, then the <OR> statement is retained.

- If a disjunct did not match in a rule trace, then its CE's are treated the same way that the CE's in a corresponding, matched disjunct were treated.

If there are no *Temporary-CE*'s and no *Precondition-CE*'s in the rule, then no iterative rule is written (since the iterative rule would be equivalent to the source rule). However, writing no iterative rule at all is still faster than writing macro-operators for specific values of N.

3.2 Efficiency of Iterative Operators

Cheng in [Cheng & Carbonell, 1986] used the source rule's left-hand-side as the precondition to the iterative-operator, so even though it was not in the iterative agenda, it incurred as much of a match cost as it did without the iterative operator. With both the source rule and iterative rule in the production set, the algebra system actually slowed down. Cheng's algorithm would only speed up a rule set if gains in eliminating false-path operators offset the overhead of matching the extra rules. Our *pre-operator* is modified so that during the execution of the iterative loop, it incurs no extra match cost. Furthermore, since the preconditions, exit condition, and side-effects of the iterative rule in our system are equivalent to the source rule, the source rule no longer has to be matched, causing a significant speed-up.

The speed of the non-iterative and the equivalent iterative operators are compared for two rule sets in figure 7. Timings of the iterative operators include the time spent firing the pre-operator and post-operator. The non-iterative and the iterative algebra macro-ops are shown in figures 6 and 8, respectively. The second example rule set controls an "organism" program in the *World Modeling System* [Carbonell & Hood, 1986]. The operator runs a simulated organism [Tallis, 1988] through one cycle of sensing its environment and performing actions in it. The "organism"

rule shows a vast improvement when only RETE match time is taken account (third column) because much of its total time is spent interacting with the world. As the third row shows, iterative operators are many times faster than non-iterative operators.

	#CE's	Total time	Match time
Original	7	*12.47 sec.*	*11.87 sec.*
Iterative	4	*3.09 sec.*	*1.76 sec.*
Original/Iterative		*4.04 x*	*6.74 x*

Algebra Rule Set, 32 equations

	#CE's	Total time	Match time
Original	9	*8.58 sec.*	*4.03 sec.*
Iterative	3	*5.03 sec.*	*0.67 sec.*
Original/Iterative		*1.71 x*	*6.01 x*

Organism Rule Set

Figure 7: Comparison of Iterative Operators with Non-Iterative

```
CONDITIONS:
 The goal is to iterate m-solve-for-unknown-n,
 There is a variable =V that isn't *Param-U*
   and that equation *Param-E1* contains,
 EITHER:
   There is another equation =E2 that is
     not *Param-E1*,  contains =V and has
     formula =F1 on its RHS,
 OR:
   There is no equation with =V on its LHS,
   But another equation =E2 contains =V
     and has formula =F2 on its RHS,
 And no equation besides *Param-E1* and =E2
   contains =V.
```

Figure 8: Conditions of the iterative rule finding the next equation to eliminate, generated by Fermi. *Param-U* and *Param-E1* are bound by the pre-operator.

3.3 Discussion

Shavlik and DeJong, in [Shavlik & DeJong, 1987], moved *all* tests out of the iterative cycle by employing E.B.L. techniques. However, this just moves the search for the appropriate working memory elements to the pre-iterative phase, and shares the same *operational/generality* trade-off [Mitchell *et al*, 1986] to which all E.B.L. systems are subject.

Transforming the preconditions to tests of the initial state only helps when work is saved. For example, in the block-unstacking problem, search for a block on which to place the next block is avoided by noting that it is the same block that was just unstacked. However, in the algebra example, a new equation and a new variable must be found on each iteration. Moving this search to the pre-iterative step will not help, and actually slows down the search.

The match in the iterative rule generated by FERMI (see figure 8) is $O(n^2)$. The search for a new variable in the variable list of equation ***Param-E1*** is $O(n^2)$, and the test for an equation is $O(n^2)$.

If an E.B.L. algorithm were used to express the tests for the next variable and next equation in terms of the initial state, then for each iteration we would need to search *all* equation variable lists to see if each variable was eligible. This is because no intermediate computations are done: the union operator performed every cycle is transformed by the domain theory into tests against all of the equations. This is an $O(n^3)$ search for each iteration.

Additional generalization is realized by examining the changes to *Temporary-CE*'s. An action can be generalized to a single operation in terms of N by using recurrence relations when the following conditions hold:

- The same operation is applied to the WME each cycle,
- the WME is not used during the iteration,
- the operation does not depend on other temporary WME's.

We are currently using a table of simple algebraic expressions to generalize in this way, but this could be extended to use more general templates as described in [Bentley *et al*, 1978]. This is more general than the original FERMI approach, which only induced an addition or subtraction of N from the *plus* or *minus* operator.

4 Conclusion and Future Work

We have added the following steps to the knowledge-compilation paradigm:

- Detection and juxtaposition of similar rule subsequences.
- Composition of disjunctive macro-operators from parallel subsequences.
- Generalization to N by composing recursive subsequences into efficient iterative macro-operators.

Since both the disjunctive and iterative composition methods compose production rules into other production rules, they are domain-independent and work independently of each other. However, when combined they yield even more dramatic improvements. For example, the algebra rule set is sped up by a factor of 7.25 over traditional composition methods.

In the future we hope to develop a better analysis of analogous rule sequences by applying our learning method to more domains. Also, while this method has proven quite effective for the problems for which we have tested it, we plan to further evaluate its usefulness by examining more domains. Reordering disjuncts based on maximizing test sharing, match cost and likelihood of early success will lend further improvements.

5 Acknowledgments

We would like to thank Patty Cheng for opening the door to a much more powerful class of macro-operators; Steve Minton for pointing out the general recurrence relation templates in [Bentley *et al*, 1978], and Klaus Gross, Angela Hickman, Ben MacLaren, and Hans Tallis for helpful comments on earlier drafts of this paper.

[Anderson, 1983] Anderson, J. A. Acquisition of Proof Skills in Geometry. In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (editors), *Machine Learning, An Artificial Intelligence Approach.* Tioga Press, Palo Alto, CA, 1983.

[Bentley *et al*, 1978] Bentley, J., Haken, D. and Saxe, J. A General Method for Solving Divide-and-Conquer Recurrences. Technical Report, Carnegie Mellon University Computer Science Department, 1978.

[Carbonell, 1983] Carbonell, J. G. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In R. S. Michalski, J. G. Carbonell and T. M. Mitchell (editors), *Machine Learning, An Artificial Intelligence Approach.* Tioga Press, Palo Alto, CA, 1983.

[Carbonell, 1986] Carbonell, J. G. Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition. In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (editors), *Machine Learning, An Artificial Intelligence Approach, Volume II.* Morgan Kaufmann, 1986.

[Carbonell & Hood, 1986] Carbonell, J. G. and Hood, G. The World Modelers Project: Learning in a Reactive Environment. In Mitchell, T. M., Carbonell, J. G. and Michalski, R. S. (editors), *Machine Learning: A Guide to Current Research*, pages 29-34. Kluwer Academic Press, 1986.

[Cheng & Carbonell, 1986] Cheng, P. W. and Carbonell, J. G. Inducing Iterative Rules from Experience: The FERMI Experiment. In *Proceedings of AAAI-86*, 1986.

[DeJong & Mooney, 1986] DeJong, G. F. and Mooney, R. Explanation-Based Learning: An Alternative View. *Machine Learning Journal*, 1(2), 1986.

[Doyle, 1984] Doyle, J. Expert Systems Without Computers. *AI Magazine*, 5(2):59-63, 1984.

[Fikes, 1971] Fikes, R. E. and Nilsson, N. J. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2:189-208, 1971.

[Forgy, 1982] Forgy, Charles L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17-37, 1982.

[Hammond, 1987] Hammond, K. J. Learning and Reusing Explanations. In *Proceedings of the Third International Workshop on Machine Learning*, pages 141-147, 1987.

[Iba, 1979] Iba, G. A. Learning Disjunctive Concepts From Examples. Master's thesis, M.I.T., 1979.

[Iba, 1988] Iba, G. A Heuristic Approach to the Discovery of Macro-Operators. Technical Report, GTE Laboratories, 1988. Tech Report TR88-506.1.

[Korf, 1985] Korf, R. E. Macro-operators: A Weak Method for Learning. *Artificial Intelligence*, 26:35-77, 1985.

[Laird *et al*, 1986] Laird, J. E., Rosenbloom, P. S. and Newell, A. Chunking in SOAR: The Anatomy of a General Learning Mechanism. *Machine Learning*, 1(1):11-46, 1986.

[Minton, 1985] Minton, S. Selectively Generalizing Plans for Problem Solving. *Proceedings of AAAI-85*, 1:596-599, 1985.

[Minton & Carbonell, 1987] Minton, S. N. and Carbonell, J. G. Strategies for Learning Search Control Rules: An Explanation-Based Approach. In *Proceedings of IJCAI-87*, Milan, Italy, 1987.

[Minton *et al*, 1988] Minton, S. N., Carbonell, J. G., Knoblock, C. A. and Kuokka, D. R. Explanation-Based Learning: Improving Problem Solving Performance through Experience. In Carbonell, J. (editor), *Paradigms for Machine Learning*. (to appear), 1988.

[Mitchell *et al*, 1986] Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T. Explanation-Based Generalization: A Unifying View. *Machine Learning*, 1:47-80, 1986.

[Newell, 1981] Newell, A. and Rosenbloom, P. Mechanisms of Skill Acquisition and the Law of Practice. In J. R. Anderson (editor), *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Erlbaum Assoc., 1981.

[Quinlan, 1986] Quinlan, J. R. Induction of Decision Trees. *Machine Learning*, 1:81-106, 1986.

[Riddle, 1988] Riddle, Patricia J. An Approach for Learning Problem Reduction Schemas and Iterative Macro-operators. In *Proceedings of the First International Workshop in Change of Representation and Inductive Bias*, 1988.

[Sankoff, 1983] Sankoff, David and Kruskal, Joseph B. An Anthology of Algorithms and Concepts for Sequence Comparison. In David Sankoff and Joseph B. Kruskal (editors), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 265-310. Addison-Wesley, 1983.

[Schank, 1982] Schank, R. C. *Dynamic Memory*. Cambridge University Press, 1982.

[Schank, 1983] Schank, R. C. The Current State of AI: One Man's Opinion. *AI Magazine*, IV(1):1-8, 1983.

[Shavlik & DeJong, 1987] Shavlik, F. W. and DeJong, G. F. An Explanation-Based Approach to Generalizing Number. In *Proceedings of IJCAI-87*, 1987.

[Shell & Carbonell, 1986] Shell, P. and Carbonell, J. G. The FRuleKit Reference Manual. 1986.CMU Computer Science Department internal paper.

[Tallis, 1988] Tallis, H. Tuning Rule-Based Systems to Their Environments. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 8-14, 1988.

[Tambe, 1988] Tambe, M. and Newell, A. Some Chunks are Expensive. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 451-458, 1988.

[Wulf, 1975] Wulf, William A. *The Design of an Optimizing Compiler*. American Elsevier Pub. Co., 1975.