ACP: Reason Maintenance and Inference Control for Constraint Propagation over Intervals

Walter Hamscher
Price Waterhouse Technology Centre
68 Willow Rd, Menlo Park, CA 94025

Abstract

ACP is a fully implemented constraint propagation system that computes numeric intervals for variables [Davis, 1987] along with an ATMS label [de Kleer, 1986a] for each such interval. The system is built within a "focused" ATMS architecture [Forbus and de Kleer, 1988, Dressler and Farquhar, 1989] and incorporates a variety of techniques to improve efficiency.

Motivation and Overview

ACP is part of the model-based financial analysis system CROSBY [Hamscher, 1990]. Financial reasoning is an appropriate domain for constraint-based representation and reasoning approaches [Bouwman, 1983, Dhar and Croker, 1988]. For the most part CROSBY uses ACP in the traditional way: to determine the consistency of sets of variable bindings and to compute values for unknown variables. For example, CROSBY might have a constraint such as

$$\text{Days.Sales.in.Inventory} = \frac{30 \times \text{Monthly.Cost.of.Goods.Sold}}{\text{Average.Inventory}}$$

Given the values Average.Inventory $\in (199, 201)$ and Cost.of.Goods.Sold $\in (19, 21)$, ACP would compute Days.Sales.in.Inventory $\in (2.84, 3.02)$. Had the fact that Days.Sales.in.Inventory $\in (3.5, 3.75)$ been previously recorded, a conflict would now be recorded.

For the purposes of this paper, all the reader need know about CROSBY is that it must construct, manipulate, and compare many combinations of underlying assumptions about the ranges of variables. Contradictions among small sets of assumptions are common. This motivates the need for recording the combinations of underlying assumptions on which each variable value depends, which in turn motivates the use of an ATMS architecture to record such information.

Although there is extensive literature on the interval propagation aspects of the problem, little of the work addresses the difficulties that arise when dependencies must be recorded. The problems that arise and the solutions incorporated into ACP are:

- Since variable values are intervals, some derived values may subsume weaker (superset) interval values.

  ACP marks variable values that are subsumed as inactive via a simple and general extension to ATMS justifications. Other systems that maintain dependencies while inferring interval labels either use single-context truth maintenance [Simmons, 1986, Sacks, 1987], nonmonotonic reasoning [Williams, 1986] or incorporate the semantics of numeric intervals into the ATMS itself [Dague *et al.*, 1990].

- Solving a constraint for a variable already solved for can cause redundant computation of variable bindings and unnecessary dependencies.

  ACP deals with this problem with a variety of strategies. Empirical results show that it is worthwhile to cache with each variable binding not only its ATMS label, but also the variable bindings that must also be present in any supporting environment.

- Certain solution paths for deriving variable bindings are uninteresting for some applications.

  ACP incorporates a unary "protect" operator into its constraint language to allow the user to advise the system to prune such derivation paths.

Syntax and Semantics

ACP uses standard notation as reviewed here: $[1, 2)$ denotes $\{x : 1 \leq x < 2\}$, $(-\infty, 0)$ denotes $\{x : x < 0\}$, and $[42, +\infty)$ denotes $\{x : 42 \leq x\}$. The symbols $+\infty$ and $-\infty$ are used only to denote the absence of upper and lower bounds; they cannot themselves be represented as intervals. Intervals may not appear as lower or upper bounds of other intervals, that is, $[0, (10, 20)]$ is ill formed. $(,)$ denotes the empty set.

All standard binary arithmetic operators are supported, with the result of evaluation being the smallest interval that contains all possible results of applying the operator pointwise [Davis, 1987]. For example, $[1, 2) + (1, 2]$ evaluates to $(2, 4)$. $(1, 2)/[0, 1)$ evaluates to $(1; +\infty)$, with the semicolon replacing the comma to denote an interval that includes the undefined result of division by zero.

All binary relations r evaluate to one of $\top$ or $\bot$, obeying the following rule for intervals I_1 and I_2:

$$I_1 \; r \; I_2 \leftrightarrow \exists x, y : x \; r \; y \wedge x \in I_1 \wedge y \in I_2$$

Corollary special cases include $\forall x : x\ r\ (-\infty, +\infty)$, which evaluates to $\top$, and $\forall x : x\ r\ (,)$ which evaluates to $\bot$.

Interval-valued variables can appear in expressions and hence in the result of evaluations, for example, evaluating the expression $([1,2] = [2,4] + c)$ yields $c = [-3,0]$. Appealing to the above rule for binary relations, $c = [-3,0]$ can be understood to mean $c \in [-3,0]$.

ACP has a unary "protect" operator, denoted "!", whose treatment by the evaluator will be described later.

ACP computes the binding of each variable under various sets of ATMS assumptions. Let the state of ACP be defined by a set of Horn clauses of the form $\Sigma \rightarrow \Phi$ or $\Sigma \rightarrow V = I$, where Σ is a set of proposition symbols denoting assumptions, Φ is a constraint formula, V is a real-valued variable and I an interval. The set of Horn clauses is to be closed under

$$\Sigma_1 \rightarrow V = I, \Sigma_2 \rightarrow \Phi \ \vdash\ \Sigma_1 \cup \Sigma_2 \rightarrow \Phi[I/V] \qquad (1)$$

where $\Phi[I/V]$ denotes the result of substituting interval I for V in Φ and evaluating. $\Sigma' \rightarrow V = I'$ *subsumes* $\Sigma \rightarrow V = I$ if and only if $I' \subseteq I$ and $\Sigma' \subseteq \Sigma$. Any clause subsumed by a different clause can be deleted. Let $\beta(V, \Gamma)$, *the binding of* V *in context* Γ, be the interval I that is minimal with respect to subset, such that there is a clause $\Sigma \rightarrow V = I$ with $\Sigma \subseteq \Gamma$.

Although this abstract specification is correct for ACP in a formal sense, it does not provide good intuitions about an implementation. In particular, it is a bad idea to literally delete every subsumed clause, since that can make it harder to check subsumption for newly added clauses. There is also an implicit interaction between subsumption and β that is not obvious from the above description. Hence, the remainder of this paper describes ACP mainly in terms of the actual mechanisms and ATMS data structures used by the program, instead of the abstract specification.

Reason Maintenance

A *node* may represent any relation (Φ), with a binding ($V = I$) being merely a special case. Each node has a *label* that is a set of minimal sets of supporting assumptions [de Kleer, 1986a]. The label of node N is denoted $L(N)$. In effect, each node N representing Φ along with its label represents a set of $\Sigma \rightarrow \Phi$ clauses. By the subsumption criterion above, only the minimal environments (Σ) need to be stored. In the remainder of the paper, nodes will be shown with their labels. For example, node n_0 represents the relation $(a = b + c)$, true in the empty environment $\{\}$:

$$n_0 : \quad (a = b + c) \quad \{\}$$

Nodes n_1 and n_2 bind the variables b and c, respectively, under assumptions B and C:

$$\begin{array}{ll} n_1 : & (b = (6,9)) \quad \{B\} \\ n_2 : & (c = (10,11)) \quad \{C\} \end{array}$$

Constraint propagation creates *justifications*, which will be written as clauses of the form $N_0 \leftarrow \bigwedge_i N_i$. ATMS label propagation computes the closure under:

$$(N_0 \leftarrow \bigwedge_i N_i) \bigwedge_i \Sigma_i \in L(N_i) \ \vdash\ \bigcup_i \Sigma_i \in L(N_0) \qquad (2)$$

Continuing the example, constraint propagation yields an interval value for a, creating node n_3, justified by justification j_1, and label propagation adds $\{B, C\}$ to the label of n_3:

$$\begin{array}{lll} j_1 : & n_3 \leftarrow n_0 \wedge n_1 \wedge n_2 & \\ n_3 : & (a = (16,20)) & \{B, C\} \end{array}$$

(In a naive implementation, the system might at this point try to derive values for b or c using the new value of a; this is an instance of "reflection" and methods for preventing it will be discussed later.)

A query for the binding of variable a in the environment $\{B, C\}$ – that is, $\beta(a, \{B, C\})$ – should return node n_3 and hence interval $(16,20)$.

Unique Bindings

To control constraint propagation, there should be at most one binding per variable per environment. Suppose, for example, that we get a new value for a under assumption A, denoted by node n_4:

$$n_4 : \quad (a = (17,19)) \quad \{A\}$$

Since this value of a is a subset of the interval for a derived earlier, a new justification is required for n_3, with a resulting update to the label of n_3:

$$\begin{array}{lll} j_2 : & n_3 \leftarrow n_4 & \\ n_3 : & (a = (16,20)) \quad \{B, C\}\{A\} & \text{Label update} \end{array}$$

Note that node n_3 representing the less specific interval $(16,20)$ for a will need to be kept along with its label. $\beta(a, \{B, C\})$ should still find node n_3 and return $(16,20)$, but $\beta(a, \{A\})$ should only find node n_4, even though n_3 is true as well. "Shadowing" justifications are introduced to provide this functionality.

A shadowing justification obeys (2), that is, the consequent is true in any environment in which all its antecedents are true. This criterion results in updates to the node labels $L(N)$. However, all nodes also have a "shadow label." Any node supported by a shadowing justification in environment Σ also has Σ added to its shadow label $S(N)$, obeying the usual minimality convention. ACP distinguishes between nodes being *true* in an environment, and *active* in an environment:

$$\begin{array}{rcl} \mathit{true}(N, \Gamma) & \leftrightarrow & \exists \Sigma \in L(N) : \Sigma \subseteq \Gamma \\ \mathit{active}(N, \Gamma) & \leftrightarrow & \mathit{true}(N, \Gamma) \\ & & \wedge \neg \exists \Sigma \in S(N) : \Sigma \subseteq \Gamma \end{array} \qquad (3)$$

Intuitively, shadowing environments make the node invisible in all their superset environments. A node shadowed in the empty environment $\{\}$ would be true in all environments, but no inferences would be made from it.

The unique binding function β is thus defined in terms of the *active* predicate:

$$\beta(V,\Gamma) = I \leftrightarrow active(V=I,\Gamma) \qquad (4)$$

In the example above, j_2 would be a shadowing justification, since in any environment in which n_4 is true, n_3 should be ignored. Shadowing justifications will be denoted by clauses written with $\Leftarrow$ and the shadow label as that label appearing to the right of a "\" character. Note that any environment appearing in $S(N)$ must also be a superset of some environment in $L(N)$. However, for compactness of notation in this paper, environments that appear in both $L(N)$ and $S(N)$ will only be shown to the right of the "\" character. In the example below, the reader should understand that $L(n_3)$ is actually $\{B,C\}$ $\{A\}$:

j_2 : $n_3 \Leftarrow n_4$
n_3 : $(a = (16,20))$ $\{B,C\} \setminus \{A\}$ Label update

Since any number of different interval values for a variable can be created in any order, it is in principle possible for $O(n^2)$ shadowing justifications to be installed for a variable with n bindings. However, since shadowing is transitive some of these shadowing justifications can be deleted. For example, suppose three nodes n_{101}, n_{102}, and n_{103} are created. The sequence of new justifications and environment propagations illustrates that after j_{102} and j_{103} are created, j_{101} can be deleted:

n_{101} :	$x = [0,10]$	$\{X1\}$	
n_{102} :	$x = [4,6]$	$\{X2\}$	
j_{101} :	$n_{101} \Leftarrow n_{102}$		
n_{101} :	$x = [0,10]$	$\{X1\} \setminus \{X2\}$	Label update
n_{103} :	$x = [2,8]$	$\{X3\}$	New node
j_{102} :	$n_{103} \Leftarrow n_{102}$		
n_{103} :	$x = [4,6]$	$\{X3\} \setminus \{X2\}$	Label update
j_{103} :	$n_{101} \Leftarrow n_{103}$		
n_{101} :	$x = [0,10]$	$\{X1\} \setminus \{X2\}\{X3\}$	Label update

ACP attempts to minimize the number of shadowing justifications by deleting each one that is no longer needed. Although deleting justifications is not a normal operation for an ATMS since it can lead to incorrect labelings, this special case guarantees that $L(N)$ and $S(N)$ remain the same as if the deletion had not taken place. Since the justifications were redundant, an efficiency advantage accrues from not recomputing labels as more environments are added.

Having defined the distinction between nodes being *true* versus being *active*, we now turn to methods for controlling propagation inferences.

Propagation

ACP propagates interval values for variables using "consumers" [de Kleer, 1986b]. A consumer is essentially a closure stored with a set of nodes; it runs exactly once with those nodes as its arguments the first time they all become true. Normally, a consumer creates a new node and justification whose antecedents are the nodes whose activation triggered it. ACP is built using a focused ATMS that maintains a single consistent focus environment and only activates consumers to run on nodes that are true in that focus environment. ACP takes this focusing notion step further, running consumers only on nodes that are active.

The propagation mechanism of ACP distinguishes between constraints and bindings. A binding is a constraint of the form $V=I$. For example, $n_0 : (a = b + c)$ is a constraint and $n_1 : (b = (6,9))$ and $n_{200} : (a = 2)$ are bindings. Propagation of a constraint node works as shown in the procedure below. For simplicity, the example below shows variables bound only to integers, rather than to intervals as would be done in ACP.

1. When a constraint node becomes active, install consumers to trigger propagation on each of the variables that appear in the constraint. For example, when $n_0 : (a = b + c)$ becomes active, consumers will be installed for variables a, b, and c.
2. When a binding node for a variable becomes active, run each of its consumers; each consumer will substitute the current binding into the constraint and evaluate it. For example, when $n_1 : b = 7$ becomes active, the constraint $n_0 : (a = b+c)$ will be evaluated given n_1, to produce a new constraint $a = 7 + c$.
3. The result of the evaluation in step 2 will fall into one of four cases:
   (a) The constant $\bot$. For example, if $(a * b = 7)$ and $a = 0$, evaluation returns $\bot$. Create a justification for the distinguished node $\bot$ from the current antecedent nodes, which will result in an ATMS conflict.
   (b) The constant $\top$. For example, if $(a*b = 0)$ and $a = 0$ then the evaluation will return $\top$. Do nothing.
   (c) A binding. For example, if $a = 2$ and $a = b + 2$ then evaluation returns the binding $b = 0$. Create a new node containing the binding and justify it with the current antecedents.
   (d) A constraint. For example, if $a = 2$ and $a = b + c$ then evaluation returns $2 = b + c$. Go back to step 1 above for the new constraint.

Protection

In the expression $(a = !(b + c))$ with a, b, and c being variables, b and c are said to be protected. The effect of protection is that evaluating any expression, all of whose variables are protected, yields $\top$. For example, evaluating $([1,2] = [2,4]+!c)$ yields $\top$. In step 3(c) above, if $a = 2$ and $a = !(b + c)$ the evaluation returns $\top$, because all the variables in $2 = !(b+c)$ are protected.

The benefit of the protect operator is that the ACP user can advise the system not to ever waste effort trying to solve for certain variables. For example, CROSBY constructs linear regression equations of the form $y = \alpha_0 x_0 + \ldots + \alpha_n x_n + \beta$, with α_i and β denoting constants. In this context it makes no sense to try to

use the dependent variable y to solve for any of the n independent x_i variables. Protecting the x_i variables is a simple, local, modular way to prevent ACP from doing so.

Solution Trees

The propagation procedure above is straightforward but would in general result in unnecessary work. For one thing, given $a = b + c$, $b = 2$ and $c = 2$, it would derive $a = 4$ in two different ways. To prevent this the variables should only be bound in some strict global order (alphabetic, for example). Furthermore, subexpressions that contain operators with idempotent elements do not always require all variables to be bound before evaluating to a constant; for example, the constraint $a = b * c$, evaluated with $c = 0$, should immediately yield $a = 0$, instead of waiting for a value for b. Finally, protected variables guarantee that certain sequences of bindings and evaluations will never yield any new bindings. Although relatively minor from a purely constraint processing point of view, these are all genuine concerns in ACP because the computational overhead of creating new nodes, justifications, and consumers far outweighs the work involved in actually evaluating the constraints and performing the associated arithmetic operations.

Whenever a new constraint node is created, ACP performs a static analysis to find all the legal sequences in which its variables could be bound. The result of this analysis is represented as a directed tree whose edges each correspond to a variable in the constraint. This is called the solution tree. Each path starting from the root represents a legal sequence. The recursive algorithm for generating this tree adds a new arc for each variable appearing in the current expression, from the current root to a new tree formed from the expression derived by deleting that variable.

For example, the root of the tree for $(a = b + c)$ has three branches: one for a leading to a subtree that is the tree for $(b + c)$; one for b leading to a subtree that is the tree for $(a = c)$; one for c leading to a subtree for $(a = b)$. In this example the c branch can be pruned because $(a = b)$ is not a binding and c (alphabetically) precedes neither a nor b.

Had the expression been $(a = b * c)$, the c branch would remain because c could be bound to 0 to produce the binding $a = 0$.

Had the expression been $(!a = b + c)$, the b branch could have been pruned because the tree for the subexpression $(!a = c)$ consists only of a single branch a, which does not precede b.

Step 1 of the propagation procedure presented earlier need only install consumers on variables corresponding to branches emanating from the corresponding position in the tree. The propagator computes the solution tree once and caches it; this is worthwhile because it is not unusual in CROSBY for variables to acquire many different bindings, and it would be wasteful for ACP to repeatedly rediscover worthless sequences of variable bindings.

In an example shown earlier, recall that we had the nodes:

$$\begin{array}{lll} n_0: & (a = b + c) & \{\} \\ n_1: & (b = (6,9)) & \{B\} \\ n_2: & (c = (10,11)) & \{C\} \\ n_3: & (a = (16,20)) & \{B,C\} \end{array}$$

The solution tree ensures that the n_0 and n_1 would have been combined to get $(a = (6,9) + c)$, which would then have been combined with n_2 to get n_3, without deriving the result in the symmetric (and redundant) fashion.

Reflection

As mentioned earlier, nodes n_0 and n_3 might in principle be combined and evaluated to yield $((16,20) = b + c)$, "reflecting" back through the n_0 constraint to derive new values of b and c. In general, there is little point in attempting to derive values for a variable V using constraints or bindings that themselves depend on some binding of V.

The straightforward and complete method for checking this is to search each of the directed acyclic graphs (DAGs) of justifications supporting any binding about to be combined with a given constraint (step 2 of the propagation procedure). If that constraint appears in every DAG, inhibit propagation. Worst-case complexity analysis suggests that this method might be worth its cost. Traversing the DAG takes time linear in the number of all nodes, but maintaining shadowing justifications takes times quadratic in the number of bindings for any single variable. Since foregoing the reflection test may result in spurious bindings, the DAG strategy may pay off asymptotically. Intuitively speaking, when reflections go undetected, many extra nodes and justifications get created, and depth first searches of justification trees are fast relative to the costs associated with creating unnecessary bindings. A further improvement to this strategy might be to search the DAG only to a fixed finite depth, since binding nodes can be supported via arbitrarily long chains of shadowing justifications.

A different strategy is to cache with each node N its *Essential Support Set* $E(N)$, and test that before searching the DAG. $E(N)$ is that set of nodes that must appear in a justification DAG for any set of supporting assumptions Γ. For example, n_0, n_1 and n_2 all have empty essential support sets; node n_3 has the essential support set $\{n_0, n_1, n_2\}$. ACP tests essential support sets to see whether they contain either the constraint or any binding for the variable about to be propagated; if so the propagation is inhibited. In the example above, node n_3 will not combine with $n_0 : (a = b + c)$ (that is, the n_3 consumer that performs this computation will not run) as long as $n_0 \in E(n_3)$.

Essential support sets can be easily computed locally and incrementally each time a justification is installed, and they have the useful property that once created, they can subsequently only get smaller as propagation

proceeds. In general, when justification $N \leftarrow \bigwedge_i N_i$ is created, $E(N)$ is updated by

$$E(N)' = E(N) \cap \bigcup_i (\{N_i\} \cup E(N_i)) \qquad (5)$$

where initially $E(N)$ is represented implicitly by $\mathcal{U}$, the set of all nodes.

For example, if some new justification $n_3 \leftarrow n_{201}$ were added, nodes n_0, n_1, and n_2 could be deleted from $E(n_3)$. In that case n_3 would then appropriately continue to propagate with constraint n_0.

Essential support sets effectively cache the result of searching the support DAG ignoring ATMS labels. As compared to the complete and correct strategy, which is to search the DAG to an unbounded depth, the essential support set strategy (hereafter, ESS) can err only by not detecting reflections.

With this additional propagation machinery in place we can now follow ACP as it continues to propagate in focus environment $\{A, B, C\}$ from n_4 as shown below.

n_4 :	$(a = (17,19))$		New node
j_3 :	$n_1 \leftarrow n_0 \wedge n_2 \wedge n_4$		
n_1 :	$(b = (6,9))$	$\{B\}\{A,C\}$	Label update
j_4 :	$n_5 \leftarrow n_0 \wedge n_1 \wedge n_4$		
n_5 :	$(c = (8,13))$	$\{A,B\}$	New node
j_5 :	$n_5 \Leftarrow n_2$		
n_5 :	$(c = (8,13))$	$\{A,B\} \setminus \{C\}$	Label update

ACP creates the new node $n_5 : (c = (8,13))$, active only in $\{A, B\}$. Hence, querying ACP for the value of c yields the following results in each of the following Γ:

$$\begin{aligned} \beta(c,\{\}) = \beta(c,\{A\}) = \beta(c,\{B\}) &= (-\infty, +\infty) \\ \beta(c,\{A,B\}) &= (8,13) \quad n_5 \\ C \in \Gamma \rightarrow \beta(c,\Gamma) &= (10,11) \quad n_2 \end{aligned}$$

Overlapping Intervals

ACP needs to deal with cases in which a variable is assigned intervals which have nonempty intersections but neither is a subset of the other; these are called overlapping intervals. Suppose that nodes n_{201} and n_{202} are created with overlapping interval values $(1,10)$ and $(5,20)$. ACP creates a third node n_{203} to represent their intersection $(5,10)$, and the new node in turn shadows the two nodes that support it.

n_{201} :	$(x = (1,10))$	$\{X1\}$	
n_{202} :	$(x = (5,20))$	$\{X2\}$	
j_{200} :	$n_{203} \leftarrow n_{201} \wedge n_{202}$		
n_{203} :	$(x = (5,10))$	$\{X1,X2\}$	New node
j_{201} :	$n_{201} \Leftarrow n_{203}$		
n_{201} :	$(x = (1,10))$	$\{X1\} \setminus \{X1,X2\}$	Label update
j_{202} :	$n_{202} \Leftarrow n_{203}$		
n_{202} :	$(x = (5,20))$	$\{X2\} \setminus \{X1,X2\}$	Label update

Querying ACP for the value of x yields a different result in each of the following environments:

$$\begin{aligned} \beta(x,\{\}) &= (-\infty, +\infty) \\ \beta(x,\{X1\}) &= (1,10) \quad n_{201} \\ \beta(x,\{X2\}) &= (5,20) \quad n_{202} \\ \beta(x,\{X1,X2\}) &= (5,10) \quad n_{203} \end{aligned}$$

Although in the worst case k variable bindings could result in $O(k^2)$ overlaps, empirically the number actually created is much less than k. Intuitively, the propagation strategy ensures that overlapping intervals are only derived from the most restrictive intervals already present in the current focus environment. Whenever ACP creates a new overlapping interval, the two intervals it was created from become shadowed and trigger no more inferences in the current focus environment.

Overlapping intervals result in a small complication to reflection checking. Although in general, no information is gained by deriving a value for variable V using constraints or bindings that themselves depend on some binding of V, overlaps between intervals can in fact add information. Hence, DAG and ESS computations ignore the presence of overlap justifications.

Empirical Results

ACP is implemented in 5500 lines of ANSI Common Lisp. Roughly one half is devoted to the expression evaluator, one third is the focused ATMS, and ACP-specific code comprises the remainder. It is currently being used in a prototype program for model-based financial data analysis; the data for Figures 1 and 2 were generated using two financial models with quarterly data for a real computer peripherals manufacturer. The "small" example (data represented by ○) uses 116 variables and involves the exploration of 12 contexts. The "large" example (data represented by ◯) uses 158 variables and explores 28 contexts. The horizontal axis refers to the reflection check strategies in use: " " means the null strategy of not checking reflections; "S" refers to a *simple* strategy for which only constraints being immediately re-applied were inhibited; "D2" refers to the DAG strategy with a depth cutoff of 2, "D" refers to the DAG strategy with no depth cutoff, "E" refers to the ESS strategy, and the other columns show the use of multiple strategies as successive filters.

Figure 1: Maximum Bindings for any Variable

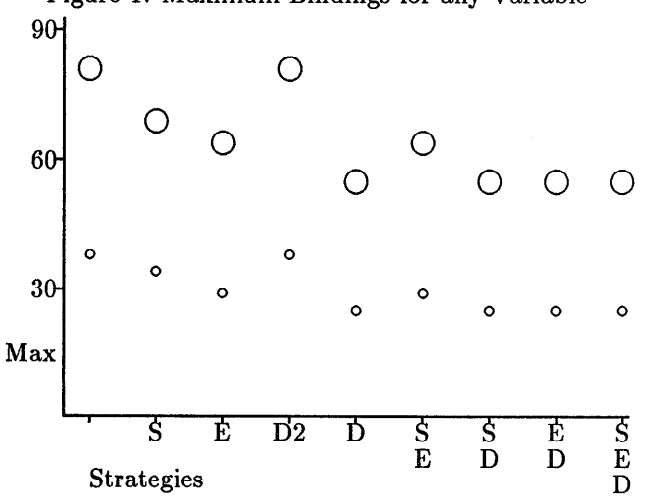

Figure 2: Run Time in Seconds

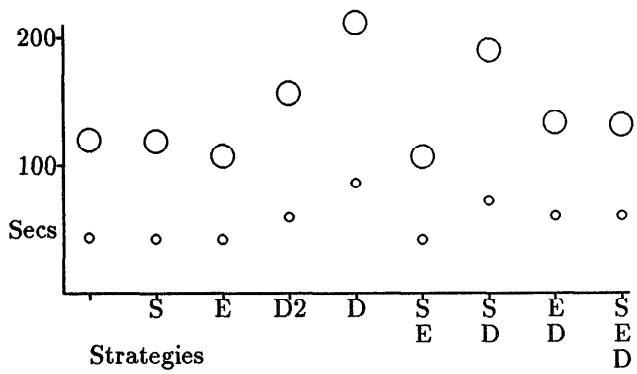

Figure 1 shows the maximum number of bindings created for any variable; in general, the stronger the reflection check strategy, the fewer intermediate results are created. Figure 1 illustrates that D2 works badly, inhibiting no intermediate bindings at all. Further, it shows ESS can work almost as well as the unlimited-depth DAG strategy.

Figure 2 shows the run time in seconds on a Symbolics 3645 (inclusive of paging and ephemeral garbage collection) for the same examples. Figure 2 demonstrates that although the worst case complexity of the DAG strategy appears superior, in fact it is empirically much more costly than ESS. Among the reasons for this are that (i) the number of shadowing justifications for a variable with k bindings is theoretically $O(k^2)$ but empirically never more than $1.5k$, and (ii) both DAG and ESS tests must be reexecuted every time the ATMS focus changes, and the frequency of such changes is larger than the average number of bindings per variable. Although ESS is not much faster than the null strategy, the space savings suggested in Figure 1 makes it the method of choice in ACP.

Conclusion

ACP integrates constraint propagation over intervals with assumption-based truth maintenance, contributing several novel inference control techniques, including the incorporation of subsumption into the ATMS and precomputing feasible solution paths for every constraint. Experiments with ACP further indicate that spurious intermediate variable bindings can be efficiently suppressed by using *essential support sets* to check whether each new variable binding is being derived in a way that depends on the variable itself.

Acknowledgements

Heinrich Taube contributed to the implementation of the expression evaluator. An anonymous reviewer suggested the inclusion of the abstract specification.

References

[Bouwman, 1983] M. J. Bouwman. Human Diagnostic Reasoning by Computer: An Illustration from Financial Analysis. *Management Science*, 29(6):653–672, June 1983.

[Dague *et al.*, 1990] P. Dague, O. Jehl, and P. Taillibert. An Interval Propagation and Conflict Recognition Engine for Diagnosing Continuous Dynamic Systems. In *Proc. Int. Workshop on Expert Systems in Engineering, in: Lecture Notes in AI*, Vienna, 1990. Springer.

[Davis, 1987] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32(3):281–332, July 1987.

[de Kleer, 1986a] J. de Kleer. An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

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

[Dhar and Croker, 1988] V. Dhar and A. Croker. Knowledge Based Decision Support in a Business: Issues and a Solution. *IEEE Expert*, pages 53–62, Spring 1988.

[Dressler and Farquhar, 1989] O. Dressler and A. Farquhar. Problem Solver Control over the ATMS. In *Proc. German Workshop on AI*, 1989.

[Forbus and de Kleer, 1988] K. D. Forbus and J. de Kleer. Focusing the ATMS. In *Proc. 7th National Conf. on Artificial Intelligence*, pages 193–198, Minneapolis, MN, 1988.

[Hamscher, 1990] W. C. Hamscher. Explaining Unexpected Financial Results. In *Proc. AAAI Spring Symposium on Automated Abduction*, pages 96–100, March 1990. Available from the author.

[Sacks, 1987] E. Sacks. Hierarchical Reasoning about Inequalities. In *Proc. 6th National Conf. on Artificial Intelligence*, pages 649–655, Seattle, WA, August 1987.

[Simmons, 1986] R. G. Simmons. Commonsense Arithmetic Reasoning. In *Proc. 5th National Conf. on Artificial Intelligence*, Philadelphia, PA, August 1986.

[Williams, 1986] B. C. Williams. Doing Time: Putting Qualitative Reasoning on Firmer Ground. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 105–112, Philadelphia, PA, August 1986.