On Determining the Consistency of Partial Descriptions Of Trees*

Thomas L. Cornell
Seminar für Sprachwissenschaft
Kleine Wilhelmstrasse 113
72074 Tübingen
Germany
cornell@sfs.nphil.uni-tuebingen.de

February 2, 1996

Abstract

We examine the consistency problem for descriptions of trees based on remote dominance, and present a consistency-checking algorithm which is polynomial in the number of nodes in the description, despite disjunctions inherent in the theory of trees. The resulting algorithm allows for descriptions which go beyond sets of atomic formulas to allow certain types of disjunction and negation.

This paper is a revision of a paper appearing in the 1994 ACL Proceedings. Several typos and one error (noted at the ACL conference) have been corrected and both the description language and its associated sequent calculus have been slightly modified. Two appendices have been added to bring things up to date, mostly adapting recent work in constraint satisfaction over systems of time intervals [16, especially] to partial descriptions of trees. The first appendix presents an important negative result—path consistent descriptions are not minimal, contrary to a claim I made at the ACL conference. However, minimality is a stronger condition than necessary, so this result does not necessarily impact on the usefulness of the *Close* algorithm as a consistency checker. The second appendix is more positive, isolating a large subset of the description language for which consistency can be checked in $O(N^2)$ time. The appendices also present a more detailed discussion of approaches to reasoning about time intervals [1, 19] that have proven very relevant to reasoning about trees.

1 Introduction

Marcus, Hindle & Fleck [9] proposed an approach to syntactic tree structures which took the primary structural relations to be remote dominance and precedence rather than immediate dominance and precedence. A partial description of a tree structure is then a set of assertions about these relations. Recently,

*Many thanks to Dick Oehrle, Ed Stabler, Drew Moshier, Mark Johnson and Terry Langendoen for comments, discussion and encouragement during the long evolution of this project.

researchers have shown a revived interest in variants of Marcus et al.'s D-theory, most likely due to the availability of approaches and techniques developed in the study of feature structures and their underlying logics. For example, [14, 2, 13, 3] present formal treatments of many notions which [9] treated only informally and incompletely, [18, 14, 12, 13] present a growing body of work on 'quasi-trees', an extension of Tree Adjoining Grammar using dominance links, and [5] shows how to integrate D-Theory with feature logics. Furthermore, work on the psycholinguistic implications of this approach has continued apace [20, 4, 8], making all the more necessary sustained foundational work in the theory of description-based tree-building applications (parsers, generators, etc.)

This paper addresses one particular problem that arises in this approach to tree building. As with feature-structures, the essential operation here is the combination of two collections of partial information about the syntactic structure of an expression. That is, the tree-building operation becomes a description-merging operation. It may happen that the two descriptions to be combined contain contradictory information. For example one might contain the assertion that "node 7 dominates node 12" while the other claims that "node 12 precedes node 7." No tree structure can satisfy both these constraints. The operation of description combination is thus not simple set union, but, like unification, involves taking a least upper bound in a semi-lattice where lub's are not everywhere defined. The *consistency problem* is thus the problem of determining the satisfiability of the output of the description merging operation.

The proposals in [14, 5, 3] all involve solving the D-Theoretic consistency problem by using assumption-based reasoning to handle the disjunctions inherent in the theory of trees. This can lead to combinatorial explosion. But as it happens, proof techniques designed to handle general disjunctions are probably more powerful than we need; the disjunctions that arise from the theory of trees are of a restricted kind which we should be able to handle by strictly polynomial means. We will see that we can efficiently handle even richer notions of description than those in the "classical" D-Theory of [9].

2 D-Theory and Tree Theory

2.1 Description Language

We will make use of the following description language $\mathcal{L}$. Define the set of *basic relation names*, $\mathbf{R}$, as:

b	—	"below" (i.e., dominated-by)
d	—	"dominates"
e	—	"equals"
f	—	"follows" (i.e., preceded-by)
p	—	"precedes"

To form the full set of *relation names*, we take the power set $P(\mathbf{R})$. An *atomic formula* of $\mathcal{L}$ is then an element of $P(\mathbf{R})$applied to a pair of node names, and interpreted disjunctively. We will usually write the elements of $P(\mathbf{R})$ without set braces or commas. For example, the formula $de(x, y)$ is taken to mean "x dominates or equals y." The formula $fp(x, y)$ is taken to mean "x follows or precedes y." In a formula $S(x, y)$ we will often refer to S as a *constraint on* *the pair* x, y. The formula $bdefp(x,y)$ is an instance of the vacuous or universal constraint. The formula (x,y), that is, the application of the empty element of $P(\mathbf{R})$ to the pair x, y (which we will henceforth write $\Box(x,y)$ for readability), is unsatisfiable, like the empty disjunction or empty clause.

Like any power set, $P(\mathbf{R})$ supports a boolean algebra of the usual sort. The vacuous constraint $bdefp$ is the $\mathbf{1}$ element of the algebra, the empty constraint $\Box$ is the $\mathbf{0}$, and the meet, join and complement operators are just set-theoretic intersection, union and complement relative to $\mathbf{R}$.

Given the axioms we present in the next section, in particular Exhaustiveness and Exclusiveness, the following equivalences hold between expressions in the predicate calculus for $\mathcal{L}$ and expressions built up from this $P(\mathbf{R})$ algebra.

$$\begin{array}{rcl}(\forall x,y)[S_1(x,y) \vee S_2(x,y) & \Longleftrightarrow & (S_1 \cup S_2)(x,y)] \\ (\forall x,y)[S_1(x,y) \wedge S_2(x,y) & \Longleftrightarrow & (S_1 \cap S_2)(x,y)] \\ (\forall x,y)[\neg S(x,y) & \Longleftrightarrow & (\sim S)(x,y)]\end{array}$$

So, for example:

$$d(x,y) \wedge p(x,y) \iff (d \cap p)(x,y)[= \Box(x,y)]$$

$$df(x,y) \wedge fp(x,y) \iff (df \cap fp)(x,y)[= f(x,y)]$$

$$\neg fp(x,y) \iff (\sim fp)(x,y)[= bde(x,y)]$$

Formalizing our semantics slightly, we treat $R_1R_2 \ldots R_n(x,y)$ as satisfied by an interpretation function I just in case $(I(x), I(y)) \in I(R_i)$ for some $1 \le i \le n$. As noted, we intend that a constraint $R_1R_2 \ldots R_n(x,y)$ have the same semantics as the clause

$$R_1(x,y) \vee R_2(x,y) \vee \ldots \vee R_n(x,y).$$

We can also think of an interpretation for a description as a *valuation*, a function assigning to every constraint a value in its relation-list. Continuing to lean towards the world of Constraint Satisfaction Problems, we can think of a description as a CSP whose variables are the relations holding among node-pairs, and a *model* for a description as a *solution* to the corresponding CSP.

Of course, atoms of the form $e(x,y)$ are satisfiable if and only if x and y denote identical members of the domain of discourse. Atoms of the form $b(x,y)$ and $f(x,y)$ are to be considered definitionally equivalent to $d(y,x)$ and $p(y,x)$, respectively.

An *$\mathcal{L}$-description* is a finite set of atomic $\mathcal{L}$-formulas. (The notion of atomic formula here is a bit confusing, since we are considering, e.g., $bde(x,y)$ to be *syntactically* an atomic formula, even though it has the semantics of a disjunction.) If a description contains only formulas with a basic relation name, we will call it *classical*, since this is the only type of description considered in [9].

We will make the further assumption here that (a) our set of node-names is linearly ordered (alphabetically) and (b) all constraints in a description are given in *normal order form*. A formula is in normal order if the first node-argument is less than or equal to the second; the normal order form of $d(x_2, x_1)$ is thus $b(x_1, x_2)$. The definition of normal order form is given in Edinburgh Prolog syntax below. Note that converting to normal order is linear in the number of basic relation names, which is bounded from above by 5, so the operation is essentially constant time.

```
normal_order( R:(X,Y), R:(X,Y) ) :- X =< Y.
normal_order( R:(X,Y), R1:(Y,X) ) :-
      X > Y,
      inverse( R, R1 ).

inverse( [], [] ).
inverse( [b,d|L1], [b,d|L2] ) :- !, inverse( L1, L2 ).
inverse( [f,p], [f,p] ) :- !.
inverse( [X|L1], [Y|L2] ) :-
      inverse1( X, Y ),
      inverse( L1, L2 ).

inverse1( b, d ). inverse1( d, b ). inverse1( e, e ).
      inverse1( f, p ). inverse1( p, f ).
```

2.2 Axioms

We make use of the following fragment of an axiomatization of tree structures, which we will assume in the background of all that follows. We postpone a formalization until the next section.

Strictness. Dominance and precedence are strict partial orders, i.e., transitive and irreflexive relations.

Equality. Equality is reflexive, and we can freely substitute equals for equals.

Inheritance. All nodes inherit the precedence properties of their ancestors. So, for example, if x precedes y and y dominates z, then x precedes z as well.

Exhaustiveness. Every pair of nodes in a tree stand in at least one of the five possible relations. I.e., x dominates y, y dominates x, x equals y, x precedes y or y precedes x, for all x and y.

A number of familiar properties of trees follow from the above system as theorems.

Exclusiveness. Every pair of nodes is related in *at most* one way. (By cases, using Inheritance, Transitivity, Substitution of equals and Exhaustiveness.)

Convergence. If two nodes both dominate a third, then they must lie on the same path to the root. (Prove by Inheritance that x precedes y and y precedes x both lead to contradictions.)

Upwards Inheritance. If x dominates y, and y precedes (resp. follows) z, then either x dominates z or x precedes (resp. follows) z. (Reductio, by cases.)

Transitivity of Equals. If x equals y and y equals z, then x equals z.

Asymmetry of Dominance and Precedence. For no x, y does x dominate y and y dominate x, and the same goes for precedence. (Note that this holds even if x and y are identical.)

This system is not meant to be an axiomatic definition of trees; it lacks a Rootedness condition, and it allows infinite and densely ordered structures, since it lacks an immediate dominance predicate and hence a way to assure that dominance is the transitive closure of immediate dominance. It is specifically adapted to the *satisfiability* problem, rather than the *validity* problem. It is relatively straightforward to show that, from any finite (atomic) description satisfying these conditions, we can construct a finite tree or a precedence ordered finite forest of finite trees (which can be extended to a finite tree by the addition of a root node). So this system is complete as far as satisfiability is concerned.

Proposition 1 *An $\mathcal{L}$-description is consistent with all of the above axioms iff it is satisfiable by a tree.*

Proof (Sketch). ($\Rightarrow$) We can (1) construct a new description over the quotient node-space modulo e; (2) list the dominance chains; (3) add a root if necessary; (4) noting that the dominance minimal elements under the root must be totally precedence ordered (they must be ordered and they cannot be dominance ordered or they would not be minimal), we number them accordingly, starting with 1; (5) apply the same procedure to the dominance ideals generated by each of the root's daughters. From the resulting numbering we can construct a "tree domain" straightforwardly. The Inheritance property assures us that dominance chains are non-tangled, so that the dominance filter generated by any node will be disjoint from the filter generated by any node precedence-ordered with respect to the first. Therefore no node will receive two addresses, and, by Exhaustiveness, every node will receive an address.

($\Leftarrow$) Trivial: the $\mathcal{L}$-theory of a tree clearly satisfies these axioms.

2.3 Deduction with Descriptions

There is a strong formal parallel among the axioms of Transitivity, Substitution of Equals, and Inheritance: each allows us to reason from a pair of basic formulas to a single basic formula. Thus they allow us to reason from classical descriptions to (slightly larger) classical descriptions. Let us refer to these axioms as *generators*. The reason for adopting $\mathcal{L}$ as a description language, rather than the simpler language of [9], is that we can now treat the Convergence property $((\forall)(d(x,y) \wedge d(z,y) \Rightarrow bde(x,z)))$ and the Upwards Inheritance property $((\forall)(d(x,y) \wedge p(y,z) \Rightarrow dp(x,z)))$ as generators. They allow us to reason from pairs of basic formulas (e.g., $d(x,y)$ and $p(y,z)$) to compound formulas (e.g., $dp(x,z)$). This means that we can express the consequences of any pair of basic formulas as a formula of $\mathcal{L}$, though possibly a compound formula. These consequences are exhibited in Figure 1. Empty cells represent the vacuous consequence $bdefp(x,y)$.

We can think of the generator table as contributing to our relation algebra a "multiplication" operator $\otimes$, with the generator table as its times table, so that the generator-consequences of $d(x,y) \wedge b(y,z)$ are just $(d \otimes b)(x,z)$, that is, $bde(x,z)$. This mulitiplication operator is almost relational composition, but not quite. For example, we write $d \otimes b = bde$, but in fact the composition of the dominance relation and its converse is a strict subset of the union of the dominance relation, its converse, and equality: $d \circ b \subset bde$.

We can generalize the notion of consequence given in the table to give us the consequences of an arbitrary pair of constraints over nodes x, y and z.

⊗	b(y,z)	d(y,z)	e(y,z)	f(y,z)	p(y,z)
b(x,y)	b(x,z)		b(x,z)	f(x,z)	p(x,z)
d(x,y)	bde(x,z)	d(x,z)	d(x,z)	df(x,z)	dp(x,z)
e(x,y)	b(x,z)	d(x,z)	e(x,z)	f(x,z)	p(x,z)
f(x,y)	bf(x,z)	f(x,z)	f(x,z)	f(x,z)	
p(x,y)	bp(x,z)	p(x,z)	p(x,z)		p(x,z)

Figure 1: Generator Table

If we think of the antecedent formulas $S_1(x, y)$ and $S_2(y, z)$ as disjunctions, then their conjunction, $S_1(x, y) \wedge S_2(y, z)$, is equivalent to the disjunction of the conjunctions of the pairs of constraints in the cross-product $S_1 \times S_2$, by Distributivity. Each disjunct is now a pair of basic formulas, and so we can apply the generator table and determine its consequences. Replacing each disjunct by its generator-consequences, we get another atomic $\mathcal{L}$-formula, a constraint on the pair x, z. So, for example,

$$\begin{array}{ll} & de(x,y) \wedge bf(y,z) \\ \Longleftrightarrow & (d(x,y) \wedge b(y,z)) \vee (d(x,y) \wedge f(y,z)) \\ & \vee\, (e(x,y) \wedge b(y,z)) \vee (e(x,y) \wedge f(y,z)) \\ \Rightarrow & (d \otimes b)(x,z) \vee (d \otimes f)(x,z) \\ & \vee\, (e \otimes b)(x,z) \vee (e \otimes f)(x,z) \\ \Longleftrightarrow & bde(x,z) \vee df(x,z) \vee b(x,z) \vee f(x,z) \\ \Longleftrightarrow & (bde \cup df \cup b \cup f)(x,z) \\ \Longleftrightarrow & bdef(x,z). \end{array}$$

This gives us the following generalization of the generators.

Generalized Generator Rule

$$\begin{array}{l} P_1 \ldots P_m(x,y) \wedge Q_1 \ldots Q_n(y,z) \\ \qquad \Rightarrow (\bigvee_{ij} P_i \otimes Q_j)(x,z) \end{array}$$

We will sometimes overload our notation slightly, and make use of $\otimes$ to stand for the effects of the Generalized Generator Rule. Such uses will be noted as they occur. In such cases, the right hand side of the above implication would be written simply

$$(P_1 \ldots P_m \otimes Q_1 \ldots Q_n)(x,z).$$

For doing formal deductions we will employ a sequent calculus adapted to our description language $\mathcal{L}$. We assume that sequents are pairs of finite sets of formulas, and we can make the further restriction that formulas on the right of the sequent arrow ("succedents") contain at most a single member. The axioms of the calculus we employ are exhibited in Figure 2, and the connective rules in Figure 3.

We have the structural rules (Weakening, Contraction, Permutation) by virtue of our use of sets and the form of the axioms. We need make no use of the Cut rule. In the presentation of the connective rules, we see them organized in matched pairs for each connective, as is standard, but in fact we consider the connectives part of the meta-language. So, for example, instead of introducing the $\wedge$ operator in the succeedent in the R$\wedge$ rule, we actually

$$\Gamma, A \rightarrow A$$

Structural Axioms

$$\Gamma \rightarrow e(x,x)$$

Reflexivity of e

$$\Gamma \rightarrow bdefp(x,y)$$
(for all x, y)

Exhaustiveness

Figure 2: D-Theory Axiom Schemata.

$$\frac{\Gamma, S_1(x,y) \rightarrow \Delta}{\Gamma, S_2(x,y) \rightarrow \Delta}\ L\cap\ \ (S_2 \subseteq S_1) \qquad \frac{\Gamma \rightarrow S_1(x,y) \quad \Gamma \rightarrow S_2(x,y)}{\Gamma \rightarrow (S_1 \cap S_2)(x,y)}\ R\cap$$

$$\frac{\Gamma, S_1(x,y) \rightarrow \Delta \quad \Gamma, S_2(x,y) \rightarrow \Delta}{\Gamma, (S_1 \cup S_2)(x,y) \rightarrow \Delta}\ L\cup \qquad \frac{\Gamma \rightarrow S_1(x,y)}{\Gamma \rightarrow S_2(x,y)}\ R\cup\ \ (S_1 \subseteq S_2)$$

$$\frac{\Gamma \rightarrow S(x,y)}{\Gamma, (\sim S)(x,y) \rightarrow}\ L\sim \qquad \frac{\Gamma, S(x,y) \rightarrow}{\Gamma \rightarrow (\sim S)(x,y)}\ R\sim$$

$$\frac{\Gamma, S_1(x,y), S_2(y,z) \rightarrow \Delta}{\Gamma, (S_1 \otimes S_2)(x,z) \rightarrow \Delta}\ L\otimes \qquad \frac{\Gamma \rightarrow S_1(x,y) \quad \Gamma \rightarrow S_2(y,z)}{\Gamma \rightarrow (S_1 \otimes S_2)(x,z)}\ R\otimes$$

Figure 3: D-Theory Inference Rules.

perform the $P(\mathbf{R})$-algebra meet operation on the succeedents of the two upper sequents.

A sequent $[\Gamma \rightarrow \Delta]$ is interpreted as an implication from conjunctions to disjunctions: if everything in Γ is true, then something in Δ must be true. It follows that $[\ \rightarrow \Delta]$ is invariably true, and $[\Gamma \rightarrow\ ]$ is invariably false. A sequent calculus proof is a tree (written right side up, with its root on the bottom) labeled with sequents. The theorem to be proved labels its root, the leaves are labeled with axioms, and all the local subtrees must be accepted by some inference rule. A proof that a description Γ_0 is inconsistent is a proof of the sequent $[\Gamma_0 \rightarrow]$. Note that

$$\frac{\Gamma \rightarrow \Box(x,y)}{\Gamma \rightarrow}$$

is a valid inference, essentially since $\Box(x,y)$ and the empty succedent both express the empty disjunction.

3 Two Algorithms

Suppose we are given an input description Γ_0 to check for satisfiability. If it is unsatisfiable, then it contradicts one of the axioms of the tree theory: Exhaustiveness, Reflexivity of Equals, Irreflexivity of Dominance and Precedence, and the Generators. A complete consistency checker must be able to exhaust the consequences of these axioms for Γ_0, monitoring for the appearance of the false formula $\Box(x,y)$.

Both algorithms take as input a description and an integer indicating the number of node-names constrained by the description. The consistency checking algorithm works as follows. The input description is assumed to be sorted according to the lexicographic ordering of its node-pairs. That is, $fp(1,2)$ would be sorted before $b(1,3)$. And as noted constraints are assumed to be in *normal order*, so that the first node argument is lexicographically less than the second.

The first step in processing is then to construct a *filter* description of the appropriate size. This description, F_N (for N the number of nodes), consists of constraints of the form $e(i,j)$, for all $i = j \leq N$, and $bdefp(i,j)$, otherwise. We then construct the pointwise meet of F_N with the input description (denoted $\Gamma \sqcap F_N$). If the input contains a violation either the Reflexivity of Equals, or the Irreflexivity of Dominance and Precedence, or of Exhaustiveness, then this pointwise meet will contain an empty constraint, and processing halts immediately. This has the extra side effect that any pair of nodes x and y not explicitly constrained in the input will now be explicitly constrained.

3.1 *Extend*: Top-Down Backtracking Search for Consistent Maximal Extensions

Given that we have begun by taking $\Gamma_0 \sqcap F_N$, for an input description Γ_0, we have only the generators left to check. The generator table defines a function from pairs of atomic formulas to consequences; to use it, we must first have atomic formulas. (In this algorithm we use only the primitive notion of $\otimes$, as defined in the generator table in Figure 1.)

Def'n 1 *A description Γ' is a* maximal extension *of a description Γ if, for every formula $S(x,y)$ in $\Gamma \sqcap F_N$, Γ' contains $s(x,y)$ for some $s \in S$.*

$$\dfrac{\dfrac{\dfrac{\Gamma \Rightarrow d(1,2) \quad \Gamma \Rightarrow p(2,3)}{\Gamma \Rightarrow dp(1,3)}\,R\otimes \qquad \Gamma \Rightarrow e(1,3)}{\Gamma \Rightarrow \Box(1,3)}\,R\wedge}{\Gamma \Rightarrow}\ \textit{see p. } 8$$

Figure 4: $\Gamma = \{d(1,2), e(1,3), p(2,3)\}$ is inconsistent.

Keep in mind that our implementation of the pointwise meet operation will "fill in" values for any constraint the user has left implicit, so a maximal extension will contain an atomic constraint for every pair of nodes in the input description.

An obvious consistency checking strategy is thus going to be to enumerate the maximal extensions of the input and feed them to the generators. If any such extension passes the generators, then it is satisfiable, and therefore it is a witness to the satisfiability of Γ_0. If the extension is unsatisfiable, then it must violate at least one of the generators. Because a maximal extension is a total assignment of node-pairs to relations, a single application of a well-chosen generator will suffice to derive a contradiction. And so a single pass through the complete set of applicable generators is sufficient to decide if a given maximal extension is consistent.

Thus, if the input description Γ_0 is inconsistent, then there is a proof of $\Gamma_0 \rightarrow$ in which every branch of the proof ends in a sub-proof like that in Figure 4. There we have the simple description $d(1,2), e(1,3), p(2,3)$, from which we can deduce $dp(1,3)$ via one of the generators (Upwards Inheritance), and $e(1,3)$, as an instance of an axiom, i.e., an instance of $\Gamma, A \rightarrow A$. Combining these with an invocation of $R\wedge$ we get the false formula $\Box(1,3)$.

The roots of these sub-proofs can be combined using $L\vee$ until we eventually build up the input description on the left of the sequent arrow, thus proving Γ_0, for non-classical Γ_0, as in Figure 5.

The following fragment of a Prolog implementation of max_extension/3 can be seen as implementing a backwards chaining search for such a "$R\wedge - L\vee$" proof. The input to both *Extend*, and to *Close* (see next section), is assumed to be an $\mathcal{L}$-description together with an integer giving the number of node-names subject to the description. The node-count is used to construct the appropriate F_N for this description. Note that, aside from implementing pointwise $\sqcap$, merge_descs/3 checks for the derivation of an empty constraint, and fails if that happens.

The algorithm's main control structure is a recursion on an agenda, a list of constraints whose effects are to be propagated through the current description. Initially the agenda is just a copy of the input description. Later, if the processing of a constraint on the agenda causes the tightening of a constraint in the description, the newly narrowed constraint is added to the agenda. Note that every constraint appears on the agenda at least once and at most five times, since any given constraint can be narrowed at most five times before it becomes empty. Therefore, although the code makes it look like the agenda is constantly growing, in fact its length is limited by the number of constraints in the input description, and so the algorithm is guaranteed to terminate after at most $5N$ steps, where N is the number of constraints in the description, and a 'step',

$$\dfrac{\dfrac{\text{Figure 4}}{d(1,2), e(1,3), p(2,3) \Rightarrow} \quad \dfrac{\text{Similarly}}{p(1,2), e(1,3), p(2,3) \Rightarrow}}{\underset{(1)}{dp(1,2), e(1,3), p(2,3) \Rightarrow}} \; L\vee$$

$$\dfrac{\dfrac{\text{Similarly}}{d(1,2), f(1,3), p(2,3) \Rightarrow} \quad \dfrac{\text{Similarly}}{p(1,2), f(1,3), p(2,3) \Rightarrow}}{\underset{(2)}{dp(1,2), f(1,3), p(2,3) \Rightarrow}} \; L\vee$$

$$\dfrac{\overset{(1)}{dp(1,2), e(1,3), p(2,3) \Rightarrow} \quad \overset{(2)}{dp(1,2), f(1,3), p(2,3) \Rightarrow}}{dp(1,2), ef(1,3), p(2,3) \Rightarrow} \; L\vee$$

Figure 5: $\Gamma = \{dp(1,2), ef(1,3), p(2,3)\}$ is inconsistent. (A "$R\wedge$-$L\vee$" proof.)

here, is the determination of the consequences of the currently active agenda item.

```
max_extension( D0, N, Extension ) :-
        filter( N, Filter_N ),
        merge_descs( D0, Filter_N, D1 ),
        Agenda = D1,
        extend( Agenda, D1, Extension ).

extend( [], X, X ).
extend( [C0|Cs], D0, X ) :-
        consequences( C0, D0, Conseqs ),
        meet_rule( Conseqs, D0, D1, NewCons ),
        merge_descs( NewCons, Cs, Agenda1 ),
        extend( Agenda1, D1, X ).
```

Meet_rule/4 and merge_descs/3 both compute the pointwise meet of two descriptions. Meet_rule/4 differs from merge_descs/3 in (a) sorting its first argument and (b) deriving both the merged description ($D1$) and a list of those consequences which actually had an effect on $D0$. Both merge_descs/3 and meet_rule/4 are based on routines for ordered set union from [11]. The main difference is that ordering is defined on the node-pairs of the constraint, rather than on the term expressing the constraint as a whole; equality is defined so that two constraints are 'equal' if they constrain the same node pair, and if two formulas are equal in this sense, then the output contains the boolean meet of their respective relation name expressions. The truly new consequences derived by meet_rule/4 are then added to the remaining agenda (Cs) with another call to merge_descs/3. (If $NewCons$ were merely appended to Cs, we could have two constraints on the same pair of nodes in the agenda at once, either of which may be less tightly constrained than the result of merging the two instances.)

The core of extend/3 is consequences/3, which determines for any given constraint what consequences it has when paired with each of the constraints in the description. Consequences/3 has two clauses; the first handles compound formulas, while the second handles atomic formulas. The first clause of consequences/3 invokes the Splitting Rule, which implements the $L\vee$ rule. The second clause of consequences/3 invokes the Generator Rule, i.e., generator lookup.

Note that, instead of exhausting the consequences of the Splitting Rule and then applying the Generator Rule (so as to discover a true $R\wedge -L\vee$ proof), we apply the Generator Rule whenever we can.[1] This is because it can act to prune away options from its consequents, thus minimizing the combinatorial explosion lurking behind the Splitting Rule. The same generator can rule out many different maximal extensions. This way, if an application of the Generator Rule does lead to the discovery of an inconsistency, then the program immediately backtracks to its last application of the Splitting Rule, pruning away from its search tree all further consequences of its inconsistent choice.

```
consequences( Constraint, Description, Consequences ) :-
        compound_formula( Constraint ),
        splitting_rule( Constraint, Consequences ).
consequences( Constraint, Description, Consequences ) :-
        atomic_formula( Constraint ),
        generator_rule( Description, Constraint, Consequences ).

atomic_formula( [_]:(_,_) ).
compound_formula( [_,_|_]:(_,_) ).

splitting_rule( Constraint, Assumption ) :-
        Constraint = Rels:Nodes,
        member( R, Rels ),
        Assumption = [R]:Nodes.
```

The Generator Rule, implemented here as generator_rule/3, scans the current description for formulas which form a "connected pair" with *Constraint*. Observe that in all our examples so far, we have carefully (and silently!)presented inputs to the generators as $s_1(x,y), s_2(y,z)$. Such a combination can be looked up directly in the generator table. However, $s_1(x,y), s_2(z,y)$ is no less a connected pair. In order to use these constraints as indices into the generator table, however, we need to invert the second member, giving $s_2^{-1}(y,z)$. This is done by connected_order/4, which returns the connected form of the formulas—if they have a connected form—or else fails. If it succeeds, then there is an entry in the generator table which gives the consequences of that connected pair. This consequence (XZ) is then placed in normal order $(C3)$, and added to the output list of consequences.

If $C2$ is not connected to $C1$, or if it is a compound formula, then it is skipped. Skipping compound formulas does not affect the completeness of the algorithm. All elements in the description (including the ignored compound

[1] It is probably even more efficient to solve *all* of the Generator Rule consequences before making *any* use of the Splitting Rule. In fact, in general, the most efficient strategy would be to determine the consequences of constraints proceeding from narrowest to broadest. See [6] for more ideas on how such an algorithm can be sharpened up.

formula) are also present on the agenda. Therefore the ignored compound formula will eventually appear at the top of the agenda and be subjected to the Splitting Rule, the result being placed on the agenda. It will then eventually be paired with $C1$'s entry in the description by the Generator Rule. The only difference will be in which formula is the left antecedent and which the right; but that does not matter, since they will be converted to connected form in any case, and their result will be converted to normal order.[2]

```
generator_rule( [], _C, [] ).
generator_rule( [C2|Rest], C1, [C3|Conseqs] ) :-
        atomic_formula( C2 ),
        connected_order( C1, C2, XY, YZ ),
        gen( XY, YZ, XZ ),
        normal_order( XZ, C3 ),
        generator_rule( Rest, C1, Conseqs ).
generator_rule( [C2|Rest], C1, Conseqs ) :-
        atomic_formula( C2 ),
        \+ connected_order( C1, C2, _, _ ),
        generator_rule( Rest, C1, Conseqs ).
generator_rule( [C2|Rest], C1, Conseqs ) :-
        compound_formula( C2 ),
        generator_rule( Rest, C1, Conseqs ).

connected_order( P:(A,B), Q:(B,C), P:(A,B), Q:(B,C) ).
connected_order( P0:(B,A), Q:(B,C), P:(A,B), Q:(B,C) ) :-
        inverse( P0, P ).
connected_order( P:(A,B), Q0:(C,B), P:(A,B), Q:(B,C) ) :-
        inverse( Q0, Q ).
connected_order( P0:(B,A), Q0:(C,B), P:(A,B), Q:(B,C) ) :-
        inverse( P0, P ),
        inverse( Q0, Q ).
```

Every rule applied in this procedure is based on a rule in the associated sequent calculus. The Splitting Rule is just $L\vee$; the Meet Rule is $R\wedge$; and the Generator Rule is just $R\otimes$. So there can be little doubt that the algorithm is a sound implementation of a search for a "$L\vee$-$R\wedge$" proof of $\Gamma_0 \rightarrow$. That it is complete follows from the fact that consistent maximal extensions are Hintikka sets. In particular, every application of a generator $\Gamma, A, B \rightarrow C$ has the same truth conditions as the set of formulas $\Gamma \cup \{\neg A\} \cup \{\neg B\} \cup \{C\}$. So a maximal extension is a Hintikka set if it contains either $\neg A$ or $\neg B$ or C for every generator instance. The exhaustiveness of our search assures this: every pair of constraints is checked at least once to see if it matches a generator. If it does not then the extension must contain either $\neg A$ or $\neg B$. If it does, then the extension contains A and B, and so it must also contain C, or be found inconsistent by the Meet Rule, alias $R\wedge$.

[2] In fact, every connected pair has two connected forms: $S_1(x,y), S_2(y,z)$ and $S_2^{-1}(z,y), S_1^{-1}(y,x)$. Unsurprisingly, in this case the output of the generator table for $S_2^{-1}(z,y)$ and $S_1^{-1}(y,x)$ will be the inverse of what it is for S_1 and S_2. In either case, the output will be placed in normal order before being entered into the description, so we have the required commutativity.

However, completeness is purchased at the cost of the complexities of exhaustive search. Note that the Splitting Rule is the only source of non-determinism in the program. All of the routines whose definitions were left out are deterministic. The ordered set manipulations are linear in the size of the combined input lists; the sort called by the Meet Rule is just a variant of merge-sort, and so of $N \log N$ complexity; the many inversions which may have to be done to switch from normal order to connected order and back again are linear in the length of the constraint list, which is bounded from above by 5, so they can be treated as constant time operations. It is only the Splitting Rule that causes us trouble. The second algorithm attempts to address this problem.

3.2 The Application of Local Consistency Techniques

We have so far treated the consistency problem for $\mathcal{L}$-descriptions as a problem in logic, with descriptions as sets of clauses of a certain restricted sort. However, as alluded to above, we can also view it as a Constraint Satisfaction Problem, in which the relation holding between a pair of nodes is a variable whose possible values are restricted to $\mathbf{R}$. The axioms define for us a pair of local consistency properties, along the lines codified in [7]. That is, the axioms of Exhaustiveness and the Reflexivity of Equals define a basic notion of Node Consistency (here "Node" refers to a variable of the CSP, not a tree-node). The generator axioms (extended to the complete generator table in Figure 1) give us a consistency property which is essentially the Path Consistency of [7], and in particular is the same as that employed in [1].

Montanari [10] defines the "central problem" in constraint satisfaction as the discovery of the *minimal* constraint network equivalent to a given input. A constraint network (description, *mutatis mutandis*) is *minimal* just in case every value of every constraint is part of at least one solution.

Def'n 2 *A description Δ is* minimal *iff every value in every constraint in Δ is part of at least one solution for Δ. (Where a solution is essentially the atomic theory of a minimal model for Δ.)*

Clearly there is only one unsatisfiable minimal description: that in which every constraint is empty. So if we can compute the minimal description corresponding to an input description, then we have determined its satisfiability (or otherwise). The problem with this rosy picture is that this central problem is in general solvable only with exponential means. However, there are cases where purely local consistency techniques, such as those outlined in [10] and [7], can assure global consistency.

In the following, we represent the constraint on a pair of nodes x and y as $[x, y]$. That is we assume for convenience the existence of a function represented as $[\cdot]$ which maps pairs of nodes to the constraint that holds of them.

In general, following [10], the best possible approximation of a minimal description is the *closure* of the input description. *Closure* means different things depending on which local consistency properties one is applying. Here we will take the Generalized Generator Rule as our consistency property. A description will thus be *closed* if it is deductively closed under the application of the Generalized Generator Rule. (See p. 6.) Equivalently:

Def'n 3 *A description* Δ *is* closed *iff, for every constraint* $S(x, y) \in \Delta$*, where we denote the value of the constraint on* x, y *in the original input as* S_0*:*

$$S = S_0 \wedge (\bigwedge_i ([x, z_i] \otimes [z_i, y])).$$

An algorithm for computing the closure of a description is thus a complete consistency checker if the following claim can be established for it: *a closed description is minimal.* This is equivalent to the claim that no constraint in a closed description contains a value which is not part of at least one solution. Or, put another way, any choice of value for any constraint can be extended to a solution for the whole system.

A minimal description is a very nice thing to have even aside from its usefulness in consistency checking. In a system in which there may be no unique most general solution to the description 'unification' problem, a minimal description is the most compact possible single representation of all of the solutions. Therefore it plays a role strongly analogous to the familiar notion of a most general unifier. However, as we might expect, a minimal description is a strictly weaker concept than that of a mgu, and the problem of extracting all the actual solutions from a minimal description remains exponentially hard in the worst case. Consider the following description.

$$\Gamma_0 = d(1,2), dp(1,3), dp(2,3)$$

Γ_0 is closed and satisfiable. However, if we examine the set of maximal extensions gotten by combining the values in Γ_0 in all possible ways, we note that one of them:

$$\Gamma_3 = d(1,2), p(1,3), d(2,3)$$

is unsatisfiable. There is nothing in Γ_0 to tell us that one combination of the values it presents is impossible. Therefore if we want to unpack a minimal description we are still left with a taxing combinatorial problem.

4 The Closure Algorithm

The basic design problem to be solved for a D-theoretic consistency checker is that the generator table accepts only atomic formulas as inputs, while the description whose consistency is at issue may contain any number of compound formulas. *Extend* solved this problem by 'bringing the description to the generators,' 'unfolding' the compound formulas into multiple maximal extensions. *Close* solves this problem by 'bringing the generators to the description,' using the Generalized Generator Rule.

Figure 6 represents a proof that $\{dp(1,2), bf(1,3), dp(2,3)\}$ is inconsistent. The initial stages of the proof involve combining the effects of the $R\otimes$ rule by means of the $L\vee$ and $R\vee$ rules until the two antecedent atoms match a pair of compound atoms found in the input description (in this case $dp(1,3)$ and $dp(2,3)$). Then this 'compound generator" is fed into the $R\wedge$ rule together with the corresponding structural axiom, generating our inconsistency.

Close, like *Extend*, implements a backwards chaining search for a proof of the relevant sort. The code for the two algorithms has been made almost identical, for the purposes of this paper. The sole essential difference is that now

$$
\dfrac{\dfrac{\dfrac{\vdots}{d(1,2), bf(1,3), d(2,3) \Rightarrow d(1,3)}\; R\otimes}{d(1,2), bf(1,3), d(2,3) \Rightarrow dp(1,3)}\; R\vee \qquad \dfrac{\dfrac{\vdots}{p(1,2), bf(1,3), d(2,3) \Rightarrow p(1,3)}\; R\otimes}{p(1,2), bf(1,3), d(2,3) \Rightarrow dp(1,3)}\; R\vee}{dp(1,2), bf(1,3), d(2,3) \Rightarrow dp(1,3)}\; L\vee
$$

(1)

$$
\dfrac{\dfrac{\vdots}{d(1,2), bf(1,3), p(2,3) \Rightarrow dp(1,3)}\; R\otimes \qquad \dfrac{\dfrac{\vdots}{p(1,2), bf(1,3), p(2,3) \Rightarrow p(1,3)}\; R\otimes}{p(1,2), bf(1,3), p(2,3) \Rightarrow dp(1,3)}\; R\vee}{dp(1,2), bf(1,3), p(2,3) \Rightarrow dp(1,3)}\; L\vee
$$

(2)

$$
\dfrac{\overset{(1)}{dp(1,2), bf(1,3), d(2,3) \Rightarrow dp(1,3)} \qquad \overset{(2)}{dp(1,2), bf(1,3), p(2,3) \Rightarrow dp(1,3)}}{dp(1,2), bf(1,3), dp(2,3) \Rightarrow dp(1,3)}\; L\vee
$$

(3)

$$
\dfrac{\overset{(3)}{dp(1,2), bf(1,3), dp(2,3) \Rightarrow dp(1,3)} \qquad dp(1,2), bf(1,3), dp(2,3) \Rightarrow bf(1,3)}{dp(1,2), bf(1,3), dp(2,3) \Rightarrow \Box(1,3)}\; R\wedge
$$

Figure 6: A $L\vee$-$R\wedge$ proof that $\{dp(1,2), bf(1,3), dp(2,3)\}$ is inconsistent

consequences/3 has only one clause, which invokes the Generalized Generator Rule. The input to apply_generator_rule/3 is the same as the input to generator_rule/3: the current description, a constraint looking to be the left antecedent of a generator, and the output consequences. Like the old rule, the new rule searches the current description for a connected formula (now not necessarily atomic). From the resulting connected pair it constructs a compound generator by taking the cross product of the atomic relations in the compound formulas (in a double loop implemented in distribute/4 and distrib_1/4), feeding the atomic pairs so constructed to the generator table (in distrib_1/4), and joining together each successive generator output. The result is a compound generator whose consequent represents the join of all the atomic generators that went into its construction.

```
apply_generator_rule( [], _C, [] ).
apply_generator_rule( [C2|Rest], C1, [C3|Cons] ) :-
        connected_order( C1, C2, S1:(X,Y), S2:(Y,Z) ),
        generator_rule( S1:(X,Y), S2:(Y,Z), C3 ),
        apply_generator_rule( Rest, C1, Cons ).
apply_generator_rule( [C2|Rest], C1, Cons ) :-
        \+ connected_order( C1, C2, _, _ ),
        apply_generator_rule( Rest, C1, Cons ).

generator_rule( S1:(X,Y), S2:(Y,Z), C3 ) :-
        distribute( S1:(X,Y), S2:(Y,Z), []:(X,Z), S3:(X,Z) ),
        normal_order( S3:(X,Z), C3 ).

distribute( []:_, _C2, Cons, Cons ).
distribute( [R1|S1]:XY, S2:YZ, S3a:XZ, S3:XZ ) :-
        distrib_1( S2:YZ, [R1]:XY, S3a:XZ, S3b:XZ ),
        distribute( S1:XY, S2:YZ, S3b:XZ, S3:XZ ).

distrib_1( []:_, _C1, Cons, Cons ).
distrib_1( [R2|S2]:YZ, S1:XY, S3a:XZ, S3:XZ ) :-
        gen( S1:XY, [R2]:YZ, S3b:XZ ),
        join( S3a, S3b, S3c ),
        distrib_1( S2:YZ, S1:XY, S3c:XZ, S3:XZ ).
```

On completion of the double loop, control works its way back to consequences/3 and thence to the Meet Rule, as in *Extend*.

Unlike *Extend*, *Close* is deterministic. Each agenda item is compared to each item in the current description, and that is that. Furthermore, the complexity of the Generalized Generator Rule is not much greater than before: the double loop we have added can only be executed a maximum of $5 * 5 = 25$ times, so we have increased the complexity of the original algorithm (considered apart from the Splitting Rule) by at most a constant factor.

Before we turn to the analysis of *Close*, however, note that its output is different from that of *Extend*. *Extend* returns a maximal extension, selected nondeterministically. *Close* returns the input description, but with values that could not be part of any solution removed.

5 Analysis of *Close*

The soundness of *Close* is easy to establish. In particular, the soundness of the Generalized Generator Rule is vouched for by the following formula of propositional logic, which is a tautology for any (finite) n.

$$((P_1 \vee \ldots \vee P_n) \wedge (P_1 \Rightarrow Q_1) \wedge \ldots \wedge (P_n \Rightarrow Q_n)) \Rightarrow (Q_1 \vee \ldots \vee Q_n)$$

It is easy to see that the Generator Rule is an instance of this tautology: each P_i represents the conjunction of a pair of atomic values in the antecedent constraints, and each minor implication represents an entry in the generator table. The disjunction on the right of the major implication corresponds to the join of the outputs of gen/3.

It is also relatively straightforward to verify that *Close* is indeed finding a $L\vee$-$R\wedge$ proof of $\Gamma_0 \Rightarrow$. Note that in our toy example of Figure 6 only a single 'compound generator' was required to derive the empty solution. In general it may take several compound generators to build a proof of $\Gamma_0 \Rightarrow$. Each one functions to remove some of the possibilities from a constraint, until eventually no possibilities are left. Thus we have a $L\vee$-$R\wedge$ proof of $\Gamma_0 \Rightarrow$ if and only if we have a proof of $\Gamma_0 \Rightarrow \Box(x, y)$, for some x and y. Let us call such a (not necessarily unique) pair a *critical pair* in the proof of $\Gamma_0 \Rightarrow$, and its associated contraint in Γ_0 a *critical constraint*.

It is not at all obvious how to choose a critical constraint beforehand, so *Close* must search for it. Every time it calls the Generalized Generator Rule and then calls the Meet Rule to merge in its consequences, it constructs a fragment of a $L\vee$-$R\wedge$ proof. We could then take the constraint which it finally succeeds in emptying out as the critical constraint, collect the proof fragments having that constraint as their succedent, and plug them together in the order they were generated to supply us with a $L\vee$-$R\wedge$ proof of $\Gamma_0 \Rightarrow$.

So *Close* will find a $L\vee$-$R\wedge$ proof of $\Gamma_0 \Rightarrow$ if one exists. It is not clear, however, that such a proof always exists when Γ_0 is unsatisfiable. *Close* is essentially a variant of the path-consistency algorithms frequently discussed in the constraint satisfaction literature [7, 1]. It is known that path-consistency is not in general a strong enough condition to ensure completeness. There are, however, special cases where path-consistency techniques are complete [10].

So far *Close* appears to be complete (two years of work have failed to turn up a counterexample), but it is unlikely to yield an easy completeness proof. The algorithm and description language presented here are strongly reminiscent of those in [1], where it is shown that the algorithm is incomplete for the temporal reasoning problems to which it was applied. However, the counterexample presented in [1] does not translate to D-theory—its closest $\mathcal{L}$-equivalent is in fact satisfiable. If *Close* is complete for D-theory, it can only be due to a substantive property of the generator axioms, that is, to properties of trees, as contrasted with properties of temporal intervals. Standard, general-purpose approachesto completeness proofs will almost certainly generalize to the temporal reasoning case.

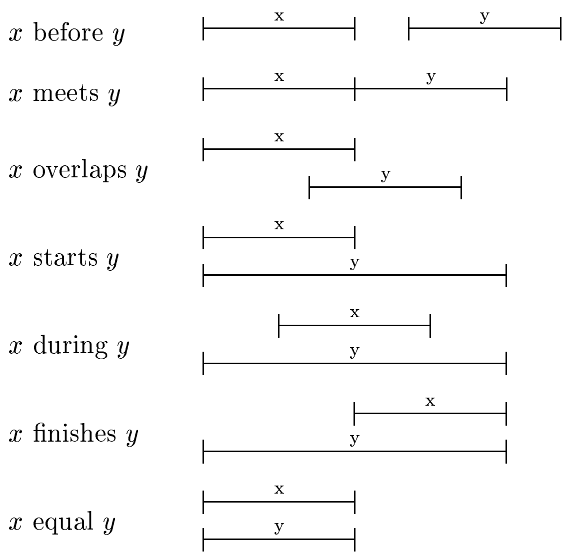

Figure 7: Basic relations of Allen's Interval Algebra IA.

Appendix A: Closed Descriptions are not Minimal

Contrary to the claim I made at the ACL meeting (and for which I thought I was actually holding a proof in my hands!), closed descriptions are *not* minimal. However, this apparently does not affect the usefulness of *Close* as a consistency checker.

The counterexample to that claim comes from work by Peter van Beek [16] on systems of temporal intervals. Indeed, most of the material in these appendices makes use of results from the study of these systems, starting with the work of James Allen [1]. Therefore it would be appropriate at this point to briefly introduce the important concepts from that literature and draw the appropriate parallels.

Allen's Interval Algebra (henceforth IA) is formally very similar to the description language $\mathcal{L}$ introduced in this paper. It consists of the primitive relations that can hold between pairs of temporal intervals: *before* (b), *meets* (m), *overlaps* (o), *starts* (s), *during* (d), *finishes* (f), and *equals* (eq), together with their inverses, denoted bi, mi, oi, etc. (Of course, eq is its own inverse.) The basic relations are illustrated in Figure 7. In addition we can form disjunctive relations in the same way as in $\mathcal{L}$, though for IA the commas and set braces are necessary for readability.

To a certain extent we can draw tight parallels between IA and $\mathcal{L}$. Suppose that we take $\mathcal{L}$ to be a language for describing the relations between "string intervals", i.e., substrings of a sentence. Then we might equate $\mathcal{L}$ dominance with the IA disjunction $\{s, d, f\}$, and $\mathcal{L}$ precedence with the IA disjunction $\{b, m\}$. The correspondence is not perfect, however, since we cannot always distinguish between the two $\mathcal{L}$ constraints $d(x, y)$ and $e(x, y)$ just by looking at the string.

That is, if we allow non-branching dominance or empty productions then two distinct nodes may cover exactly the same substring. However, van Beek's [16] counterexample to the minimality of closed descriptions carries over directly, since it can be expressed solely in terms of precedence, equality and inequality. It is as follows. Suppose we have a description like Γ, below.

$$\Gamma = \left\{ \begin{array}{lll} ep(1,2), & ep(1,3), & ep(1,4), \\ & (\sim e)(2,3), & ep(2,4), \\ & & ep(3,4), \end{array} \right\}$$

Γ is closed, but not minimal. Recall that in a minimal description, every value in every constraint is part of some solution. However, there is no tree satisfying Γ for which $e(1,4)$ is true, for in that case all the nodes collapse together into a single equivalence class, violating the inequality of nodes 2 and 4.

The good news is that Γ is consistent. Furthermore, van Beek and Cohen [17] show that up to isomorphism this is the *only* counterexample. That is, every closed constraint graph which is not minimal has a subgraph of this form. However, we cannot bring this result to bear directly on the D-theoretic consistency problem, because the result is not for IA but for a particular subset of IA, Vilain and Kautz's [19] Point Algebra (henceforth PA). Recall that IA descriptions include some which are closed but inconsistent, as noted in the main body of the paper, so *Close* cannot be a complete consistency checker for IA. D-theory falls somewhere between PA and IA: the counterexample to the adequacy of *Close* for IA, when translated into $\mathcal{L}$, actually becomes satisfiable, so D-theory is strictly weaker than IA, but there are some disjunctions that can be expressed in $\mathcal{L}$ (e.g., $fp(x,y)$) which cannot be expressed in PA. (See below for more on PA.) Therefore, alas, we still do not have a proof one way of the other for the adequacy of *Close* as a consistency checker for D-theory. However it is worth noting that even in the worst case scenario, where D-theory is shown to present counterexamples to the conjecture that *Close* is a complete consistency checker, things may not be as bad as they seem. Ladkin and Reinefeld [6] studied the use of an algorithm very much like our *Extend*, which mixes the Generator Rule and the Splitting Rule in the same way. They present empirical evidence that, for the full Interval Algebra, the pruning power of path consistency algorithms like *Close* is considerable, leaving very little room for backtracking due to the Splitting Rule. Therefore, even though *Extend* is exponential in the worst case, in practice mixing closure with splitting may yield quite usable results.

Appendix B: The Point Algebra and Tree Domains

In [19], Vilain and Kautz present a new language for expressing the relationships between time intervals based on stating constraints on the end-points of those intervals. They demonstrate that a considerable subset of the constraints expressible in IA can be expressed in this Point Algebra, PA, as well. Furthermore, van Beek [16] presents an algorithm which can be used as a consistency checker for PA descriptions with $O(N^2)$ time complexity. Therefore, while we wait for the final verdict on the adequacy of *Close* as a consistency checker for

D-theory, it seems reasonable to explore the possibility that sub-languages of $\mathcal{L}$ may yield similar results.

We have already noted and rejected the possibility of a direct translation of PA and $\mathcal{L}$: simply projecting structural relations onto the terminal string loses too much information. However, there is another alternative that proves surprisingly fruitful. The underlying goal is the same: to find a way to project the essentially two dimensional world of D-theoretic descriptions onto a single dimension, where we can reduce the consistency problem to the problem of finding loops in a graph. We do this as follows.

The concept of a tree domain is well known. We can define trees as sets of digit-strings satisfying the conditions that, for w a string and i, j digits, if (w, i) is in the set, then w is in the set, and (w, j) is in the set for all $0 < j < i$. These digit-strings then function as *node addresses*, so that $(1, 2, 1, 3)$ corresponds to the third daughter of the first daughter of the second daughter of the first daughter of the root.

Instead of treating these node addresses as digit strings, we can treat them as decimal numbers in the interval $[0, 1)$. (Of course, if our tree has nodes with more than 9 daughters, we will need to move to a number system with a larger base.) Thus the number 0.1213 corresponds as above to the third daughter of the first daughter of the second daughter of the first daughter of the root.

Now we can express the dominance and precedence relations in arithmetic terms. First, let x' stand for the address of the node denoted by x. We define a function s as follows. We can express an address n in "pseudo-scientific notation" as:

$$n = m * 10^{-k}$$

where k is the length of n, that is, the number of digits in n. Then, for $n = m * 10^{-k}$, $s(n) = (m + 1) * 10^{-k}$. Intuitively, $s(n)$ is the address of the adjacent sister of n, if n has a sister (so 's' can be read as either 'sister' or 'successor').

We can now translate dominance and precedence relations into arithmetic ordering relations.

$$\begin{array}{lcl} d(x, y) & \iff & (x' < y') \wedge (y' < s(x')) \\ p(x, y) & \iff & (s(x') \leq y') \\ e(x, y) & \iff & (x' = y') \end{array}$$

We can derive the inverse relations $b(x, y)$ and $f(x, y)$ by exchanging x's and y's. As in PA we now have a language whose relations are limited to $\{=, <, >, \leq, \geq, \neq\}$, plus the inconsistent empty relation $\emptyset$ and the vacuous constraint, which we express as '?' when we mention it at all.

Just as in PA, many of the disjunctions allowed in $\mathcal{L}$ can now be expressed as conjunctions of constraints in this new language. So, for example,

$$(\sim p)(x, y) = bdef(x, y) \iff \neg(s(x') \leq y') \iff (y' < s(x'))$$

and so, by inversion,

$$bdep(x, y) \iff (x' < s(y'))$$

and, finally, by conjunction,

$$bde(x, y) = (bdef \wedge bdep)(x, y) \iff (y' < s(x')) \wedge (x' < s(y')).$$

$$
\begin{array}{rcl}
b(x,y) & \Longleftrightarrow & (y' < x') \wedge (x' < s(y')) \\
d(x,y) & \Longleftrightarrow & (x' < y') \wedge (y' < s(x')) \\
e(x,y) & \Longleftrightarrow & (x' = y') \\
f(x,y) & \Longleftrightarrow & (s(y') \leq x') \\
p(x,y) & \Longleftrightarrow & (s(x') \leq y') \\
be(x,y) & \Longleftrightarrow & (y' \leq x') \wedge (x' < s(y')) \\
de(x,y) & \Longleftrightarrow & (x' \leq y') \wedge (y' < s(x')) \\
bdfp(x,y) & \Longleftrightarrow & (x' \neq y') \\
bdep(x,y) & \Longleftrightarrow & (x' < s(y')) \\
bdef(x,y) & \Longleftrightarrow & (y' < s(x')) \\
bdp(x,y) & \Longleftrightarrow & (x' \neq y') \wedge (x' < s(y')) \\
bdf(x,y) & \Longleftrightarrow & (x' \neq y') \wedge (y' < s(x')) \\
bde(x,y) & \Longleftrightarrow & (x' < s(y')) \wedge (y' < s(x')) \\
bd(x,y) & \Longleftrightarrow & (x' \neq y') \wedge (x' < s(y')) \wedge (y' < s(x')
\end{array}
$$

Table 1: Some correspondences between $\mathcal{L}$ and the Tree Domain Language.

Using these simple and obvious manipulations (inversion, negation and conjunction) we can derive the correspondences in Table 1.

With some effort we can add a few more formulas to this set, but first it is necessary to explore a little further the logic of this system. We can simplify our definition of the s function, using the following theorems.

$$\forall x(x < s(x)) \tag{1}$$

$$\forall x \forall y(x < y \wedge y < s(x) \Rightarrow s(y) < s(x)) \tag{2}$$

The first theorem is obviously true. The second is only true if we have chosen our base to be large enough. So, for example, if $x = .1$ and $y = .19$, and we are working in base 10, then in fact $s(y) = s(x)$. However, as long as we are careful to choose a base enough larger than the largest branching factor in the tree, then (2) will hold as well. The essential underlying concept is that the addresses for the constuent rooted in n should fall in the half-open interval $[n, s(n))$. (For most linguistic applications, we will be perfectly safe with good old base 10.) Note that because the interval is half open, we make no commitment as to the actual existence of a node with address $s(x)$ for any x. Note also that we make no use of terms like $s(s(x))$ or $s(s(s(x)))$.

Now we can add a few more formulas to our restricted description language.

$$
\begin{array}{rcll}
dp(x,y) & \Longleftrightarrow & ((x' < y') \wedge (y' < s(x'))) \vee (s(x') \leq y') & \text{[Table 1]} \\
 & \Longleftrightarrow & ((x' < y') \vee (s(x') \leq y')) \wedge ((y' < s(x')) \vee (s(x') \leq y')) & \\
 & \Longleftrightarrow & ((x' < y') \vee (s(x') \leq y')) \wedge ((y' < s(x')) \vee \neg(y' < s(x'))) & \\
 & \Longleftrightarrow & (x' < y') \vee (s(x') \leq y') & \\
 & \Longleftrightarrow & (x' < y') & \text{[(1)]}
\end{array}
$$

Immediately we also have:

$$\begin{array}{rcll} bf(x,y) & \Longleftrightarrow & (y' < x') & \text{[Inversion]} \\ bef(x,y) & \Longleftrightarrow & (y' \leq x') & [bef = \sim dp] \\ dep(x,y) & \Longleftrightarrow & (x' \leq y') & [dep = \sim bf] \end{array}$$

Finally, with some effort we can show that

$$df(x,y) \iff (s(y') < s(x')).$$

We do this in two stages. First we prove the following lemma from (2).

$$\forall x \forall y (((x < s(y)) \wedge (s(y) < s(x))) \Rightarrow (x < y)) \tag{3}$$

We refute the negation: for arbitrary x and y,

$$\begin{array}{cr} (x < s(y)) \wedge (s(y) < s(x)) \wedge \neg(x < y) & \\ (x < s(y)) \wedge (s(y) < s(x)) \wedge (y \leq x) & \\ ((y \leq x) \wedge (x < s(y))) \wedge (s(y) < s(x)) & \text{[Assoc., Commut.]} \\ (s(x) < s(y)) \wedge (s(y) < s(x)) & [(2)] \\ \square & \end{array}$$

By our definitions of d and f, we should have

$$df(x,y) \iff ((x' < y') \wedge (y' < s(x'))) \vee (s(y') \leq x').$$

By (3),

$$(s(y') < s(x')) \Rightarrow ((x' < s(y')) \Rightarrow (x' < y')).$$

So, assuming our antecedent, we have:

$$\begin{array}{c} (x' < s(y')) \Rightarrow (x' < y') \\ (s(y') \leq x') \vee (x' < y') \\ (x' < y') \vee (s(y') \leq x') \end{array}$$

Furthermore, by (1) and the transitivity of $<$,

$$(s(y') < s(x')) \Rightarrow (y' < s(x'))$$

and so, by Weakening,

$$(s(y') < s(x')) \Rightarrow ((y' < s(x')) \vee (s(y') \leq x')).$$

We conjoin our conclusions, and 'un-Distribute' $(s(y') \leq x')$, and so derive

$$(s(y') < s(x')) \Rightarrow (((x' < y') \wedge (y' < s(x'))) \vee (s(y') \leq x')).$$

We now prove the $(\Leftarrow)$ direction, that is,

$$(((x' < y') \wedge (y' < s(x'))) \vee (s(y') \leq x')) \Rightarrow (s(y') < s(x'))$$

We do this by showing that both disjuncts imply the consequent independently. First,

$$((x' < y') \wedge (y' < s(x'))) \Rightarrow (s(y' < s(x'))$$

$$
\begin{array}{rcll}
dp(x,y) & \iff & (x' < y') & \\
bf(x,y) & \iff & (y' < x') & \text{[Inversion]} \\
dep(x,y) & \iff & (x' \leq y') & \\
bef(x,y) & \iff & (y' \leq x') & \\
df(x,y) & \iff & (s(y') < s(x')) & \\
bp(x,y) & \iff & (s(x') < s(y')) & \text{[Inversion]} \\
bep(x,y) & \iff & (s(y') \leq s(x')) & [bep = \ \sim df] \\
def(x,y) & \iff & (s(x') \leq s(y')) & [def = \ \sim bp] \\
ep(x,y) & \iff & (x' \leq y') \wedge (s(x') \leq s(y')) & [ep = bep \wedge dep] \\
ef(x,y) & \iff & (y' \leq x') \wedge (s(y') \leq s(x')) &
\end{array}
$$

Table 2: More correspondences between $\mathcal{L}$ and the Tree Domain language.

$$
\begin{array}{c}
fp(x,y) \\
bfp(x,y) \\
dfp(x,y) \\
efp(x,y) \\
befp(x,y) \\
defp(x,y)
\end{array}
$$

Table 3: $\mathcal{L}$ constraints that cannot be translated.

by (2). In addition,

$$(s(y') \leq x') \Rightarrow (s(y') < s(x'))$$

by (1) and the transitivity of $<$. And so we have our proof. The solutions for dp and its immediate consequences, and for df and its immediate consequences are shown in Table 2.

This gives us equivalents for all but six $\mathcal{L}$ constraints. Those six are given in Table 3. The reader will note that all of these exceptions contain the disjunction fp. There is no way to express this disjunction conjunctively; the closest approximation we can managed is $bdfp$, which is simply '$\neq$'. This is unfortunate, since a number of linguistic frameworks attempt to factor out precedence relations, so being able to express the concept "standing in a precedence relation, but we don't care which" could be important in such a context. However, we can translate all 'classical' descriptions into this new language, and we can express all of the contents of all the cells of the Generator Table as well, as well as the weak orders de, be, ep and ef. So in many ways we lose very little in moving to this new language.

We lose very little and we gain a considerable amount. As noted, van Beek [16] has devised an algorithm which can be used as a consistency checker for this linear language.[3]

van Beek shows that descriptions of this sort are inconsistent precisely when they contain one of three types of cycles.

[3]In fact van Beek proves a stronger result: a model witnessing the consistency of the input constraints can be generated in $O(N^2)$ time. The entire algorithm is not necessary if we wish merely to check consistency, but the fragment we need is still $O(N^2)$.

1. $v = \cdots = w \neq v$

2. $v \leq \cdots \leq w \leq \cdots \leq v \neq w$

3. $v < \cdots < w < \cdots < v$

In 2, some or all of the $\leq$ can be $=$ and in 3 all but one of the $<$ can actually be either $\leq$ or $=$. van Beek shows that all three of these cycles can be detected with slight modifications to an algorithm for picking out strongly connected components in a graph. Tarjan [15] presents an efficient algorithm which visits each relation/arc only once, and so is $O(N^2)$ for N the number of terms being constrained.

To make van Beek's algorithm work for us we need to be sure that we can efficiently derive all the relations that can hold between each of the four points we introduce for every constraint in the input description. That is, for every descriptive atom $R(x,y)$, we will be introducing four new terms, $x', y', s(x'), s(y')$, and therefore ten possible new relations. Of course, four of these are trivial and immediate: $x' = x'$, etc. Two more are also immediate, consequences of (1): $x' < s(x')$ and $y' < s(y')$. Therefore we need concern ourselves only with the four relations

$$x' R_1 y', x' R_2 s(y'), s(x') R_3 y', s(x') R_4 s(y').$$

We can compute these off-line and implement the necessary inferences in a table-lookup, so translating a description from $\mathcal{L}$ to this new language should present no additional sources of complexity. However, it should be noted that I have not yet attempted to implement van Beek's algorithm as a D-theoretic consistency checker, so the reader is cautioned that at this point this is still just a Very Promising Approach.

References

[1] Allen, James F. 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM* 26(11): 832–843.

[2] Backofen, Rolf, James Rogers, & K. Vijay-Shanker. Ms. A first-order axiomatization of trees. Submitted to *Journal of Logic, Language, and Information.*

[3] Cornell, Thomas L. 1992. *Description Theory, Licensing Theory and Principle-Based Grammars and Parsers.* UCLA Ph.D. thesis.

[4] Gorrell, Paul. 1991. Subcategorization and Sentence Processing. In Berwick, R., S. Abney & C. Tenney, eds. *Principle-Based Parsing: Computation and Psycholinguistics.* Dordrecht: Kluwer.

[5] Johnson, Mark. 1994. Computing with Features as Formulae. *Computational Linguistics* 20: 1–26.

[6] Ladkin, Peter B., and Alexander Reinefeld. 1992. Effective solution of qualitative interval constraint problems. *Artificial Intelligence* 57: 105–124.

[7] Mackworth, Alan K. 1977. Consistency in Networks of Relations. *Artificial Intelligence* 8: 99–118.

[8] Marcus, Mitchell P. & Donald Hindle. 1990. Description Theory and Intonation Boundaries. In G. T. M. Altman (Ed.), *Cognitive Models of Speech Processing*, pp. 483–512. Cambridge, MA: MIT Press.

[9] Marcus, Mitchell P., Donald Hindle & Margaret M. Fleck. 1983. D-Theory: Talking about Talking about Trees. *Proceedings of the 21st Mtg. of the ACL.*

[10] Montanari, Ugo. 1974. Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences* 7: 95–132.

[11] O'Keefe, Richard A. 1990. *The Craft of Prolog.* Cambridge, MA: MIT Press.

[12] Rambow, Owen. 1994. Multiset-valued Linear Index Grammars: Imposing dominance constraints on derivations. *Proceedings of the 32nd Mtg. of the ACL.*

[13] Rogers, James. 1994. *Studies in the Logic of Trees with Applications to Grammar Formalisms.* University of Delaware Ph. D. Thesis. Dept. of Computer & Information Sciences Tech. Rept. 95-04.

[14] Rogers, James & K. Vijay-Shanker. 1992. Reasoning with Descriptions of Trees. *Proceedings of the 30th Mtg. of the ACL.*

[15] Tarjan, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM Journal of Computing* 1: 745–750.

[16] van Beek, Peter. 1992. Reasoning about qualitative temporal information. *Artificial Intelligence* 58: 297–326.

[17] van Beek, Peter, & Robin Cohen. 1990. Exact and approximate reasoning about temporal relations. *Computational Intelligence* 6: 132–144.

[18] Vijay-Shanker, K. 1992. Using descriptions of trees in a Tree Adjoining Grammar. *Computational Linguistics* 18(4): 481–518.

[19] Vilain, Marc, & Henry A. Kautz. 1986. Constraint propagation algorithms for temoral reasoning. *Proceedings AAAI-86*, Philadelphia, PA: 377–382.

[20] Weinberg, A. 1988. *Locality Principles in Syntax and in Parsing.* MIT Ph.D. dissertation.