TABLEAU-BASED THEOREM PROVING FOR A CONDITIONAL LOGIC

by

Christine Groeneboer

B.Sc., University of Wisconsin, 1974

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the School

of

Computing Science

SIMON FRASER UNIVERSITY

July 1987

Approval

Name: R. Christine Groeneboer

Degree: Master of Science

Title of Thesis: Tableau-based Theorem Proving for a Conditional Logic

Examining Committee:

Chairperson: Dr. Binay Bhattacharya

Dr. James Delgrande
Senior Supervisor

Dr. Nick Cercone

Dr. Robert Hadley

Dr. Verónica Dahl
External Examiner

August 10, 1987.
Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/~~Project~~/~~Extended Essay~~

TABLEAU-BASED THEOREM PROVING FOR A CONDITIONAL LOGIC

Author: _____
(signature)

R. CHRISTINE GROENEBOER
(name)

AUGUST 10, 1987
(date)

Abstract

This thesis presents a tableau-based approach to theorem-proving for a conditional logic. Conditional logics are those logics concerned with statements of the form "if α then β." A truth-functional semantics for the material conditional $\alpha \supset \beta$ is that $\alpha \supset \beta$ is equivalent by definition to $\neg \alpha \vee \beta$. Then whenever α is true, so is β. For example, $penguin(x) \supset bird(x)$ is true because for every x such that x is a penguin, x is a bird.

However, exception-admitting conditional knowledge such as knowledge about laws of nature, natural kinds, and properties of kinds is not readily formalized with material conditionals. Many of the properties of material conditionals fail for such "natural conditionals," e.g., transitivity: penguins are birds, birds normally fly, but it is not the case that penguins normally fly. Logic N is a conditional logic devised to represent the relation between natural kinds and properties of kinds. A variably strict conditional operator "$\Rightarrow$" is added to propositional logic. Informally, "$\alpha \Rightarrow \beta$" means "in the normal course of events, if α then β" or "all other things being equal, if α then β."

A tableau-based approach to theorem-proving for conditional logic N is presented. A tableau represents an attempt to construct a model in which $\neg\omega$ is satisfiable in order to prove ω. If such a model can be constructed, ω is invalid. Otherwise such a model is shown to be impossible to construct, and ω is therefore valid. This approach is based on semantic diagrams for modal logics and temporal logics. The method is shown to be sound and complete and has been implemented in program *Validate*, an automated theorem-prover for conditional logic N.

DEDICATION

to Rob, Jessica, and Cara

ACKNOWLEDGEMENTS

I would like to thank my senior supervisor James Delgrande for his guidance, encouragement, and support. It is greatly appreciated. Thanks to members of the committee, Nick Cercone and Robert Hadley, for discussions and comments on the thesis. I would also like to acknowledge the contribution of the external examiner, Veronica Dahl. Encouragement from and discussions with fellow graduate students, especially those in Radandt Hall, helped to make the experience a pleasant one. Thanks in particular to Pierre Massicotte for his contribution to programming. Financial support from the research grant of James Delgrande, the School of Computing Science, and Simon Fraser University is gratefully acknowledged. My deepest appreciation to family and friends for their support and understanding throughout the entire effort.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

This thesis presents a tableau-based approach to theorem proving for a conditional logical system N. Logic N extends classical propositional logic by adding a variably strict conditional operator $\Rightarrow$.[1] The general structure of the formal semantics for N is based on a possible worlds approach. The theorem prover is a refutation procedure based on the model theory of N rather than the axiomatic basis of N. An attempt is made to construct a falsifying model for a wff ω. If such a model can be constructed, then $\neg\omega$ is satisfiable and ω is invalid. Otherwise a falsifying model is shown to be impossible. In this event $\neg\omega$ is unsatisfiable, and ω is therefore valid. The *method of N-diagrams* is presented to accomplish model construction. The method consists of rectangles representing worlds, arrows representing accessibility between worlds, and rules for building models based on the definition of truth in the model theory. This approach to theorem proving is shown to be sound and complete. Program *Validate*, an automated theorem prover for N incorporating the method of N-diagrams, is presented and the computational complexity of the algorithm analyzed.

Logic N is intended to provide an approach to the representation of knowledge about prototypes and prototypical properties. Consider the statements "birds fly," "ravens are black," "water boils at 100° C." Such statements are arguably not intended to be understood as universal laws. Penguins do not fly, nor do birds with broken wings. Albino ravens are not black, and water samples with a high mineral content boil at temperatures lower than 100° C. The problem is that not every member of the kind possesses all the properties associated with the kind.

Exception-admitting knowledge poses some special representation problems. A truth functional semantics for the material conditional is given by $A \supset B \equiv_{df} \neg A \vee B$. Thus material implication does not allow exceptions. As a representation, the material conditional is fine for (1) terms which have analytic definitions such as "bachelor" or "square," (2) nouns derived by transformation from verbal forms such as "hunter = one who hunts," and (3) statements such as "penguins are birds."[2] Though conventional logics do

[1] N has been extended to full first-order in [Delgrande 87].

[2] It has been argued that perhaps material implication is too strong even for statements of type (3). Remember when whales were thought to be fishes [Brachman 85].

possess the expressive power to represent exception-allowing knowledge, the representation is not readily formulated in a straightforward manner. "Exceptions in representation systems arise as a result of (1) the sometimes unpredictable nature of the world, which produces atypical situations, and (2) the inadequacies of current representation formalisms in dealing with 'natural' concepts (as used by people)" [Lesperance 80, p.2].[3]

Interesting nondefinitional exceptionless general statements are rare. Thus automated commonsense reasoners require an exception-allowing capacity. Past approaches to this problem in AI include default logics and prototype theory. The advantage of default reasoning lies in its ability to consistently extend a set of beliefs. A prototype theory is useful if the goal is prediction of properties given an individual, or prediction of kinds given a set of properties.

Semantic problems arise with both of these approaches, though, when the goal is to represent relations between natural kinds and their attributes. As an example, consider blackness and ravenhood. Given that individual *a* is a raven and that it is not known that *a* is albino, then assume by default that *a* is black. However, the relation between blackness and ravenhood, whatever it may be, is presumably independent of a set of beliefs in a knowledge base.

1. Natural Kinds

Natural kind terms refer to a group of nouns whose members stand for naturally occurring classes such as "lemon," "bird," or "water." Past views of natural kinds have been categorized in [de Sousa 84] by the following:

(1) As essences which (a) determine that a thing is what it is, and (b) make up a necessary condition of its continued existence as that thing. Essences are to be discovered through scientific inquiry.

(2) An Empiricist or Lockean view in which the real nature of kinds is considered unknowable. Boundaries between classes of kinds are set by our concepts rather than irreducible essential properties of

[3] Lesperance distinguishes two classes of exceptional conditions: generic and individual. Generic exceptions arise when constraints in the definition of a category are violated, while individual exceptions involve violation of constraints in particular individual objects. So, for example, the property of nonflight in penguins is a generic exception to flight in birds, whereas the property of nonflight in a broken-winged raven is an individual exception.

kinds. The extension of our concepts is determined by the intension.

(3) a "modern" view in which natural kind names are perceived as rigid designators, i.e., natural kinds are selected on the basis of typical samples, and members of the same class are selected in every possible world. The goal of science is to discover the essential natures of kinds as in (1). This category includes the views of [Kripke 72] [Putnam 75] [Schwartz 79] [Salmon 81].

Procedural semantics provides rules of the form "every member of a kind has that essential nature which *usually* causes the properties associated with the kind" [Hadley 87]. Such a view has the advantage that even if our knowledge about the essential nature of a kind changes, the rules do not.

The main approaches to natural kinds representation in AI have been default reasoning and prototype theory. Reiter [78] [80] [81] [83] uses the term *default reasoning* to refer to a process of deriving conclusions based upon patterns of inference of the form "in the absence of any information to the contrary, assume..." The role of a default is seen as a means of completing the underlying incomplete first-order theory in order to allow the necessary inferences. Defaults are then instructions for creating an *extension* of this incomplete theory. An extension may be thought of as an acceptable set of beliefs about the world being represented. Different applications of the default rules may yield different extensions.

A *default* is an expression of the form

$$\frac{\alpha(x): M\beta_1(x), M\beta_2(x), ..., M\beta_m(x)}{\gamma(x)},$$

where $\alpha(x), \beta_1(x), ..., \beta_m(x), \gamma(x)$ are wffs whose free variables are among those of $x = x_1, x_2, ..., x_n$. $\alpha(x)$ is the *prerequisite* of the default, $\beta_1(x), \beta_2(x), ..., \beta_m(x)$ the *justification*, and $\gamma(x)$ the *consequent*. A default is *closed* iff none of $\alpha, \beta_1, ..., \beta_m, \gamma$ contains a free variable. A *default theory* is a pair (D,W) where D is a set of defaults, and W a set of closed wffs. A default theory is *closed* iff every default of D is closed. Intuitively, the set of defaults can be seen as extending the first-order theory W. For example, consider the following default theory about ravens:

$$W = \{ (x)raven(x) \supset bird(x) \}, \quad D = \{ \frac{raven(x): black(x)}{black(x)} \}.$$

The first-order formula in W expresses the idea that if x is a raven then it cannot help but be a bird, while

the default rule says that if it can be derived that x is a raven and it can be consistently assumed that x is black, infer that x is black. Etherington and Reiter [83] present an algorithm for reasoning with defaults. Because first-order logic is undecidable, application of the theory is restricted to a decidable subset, acyclic inheritance hierarchies. Delgrande [87] discusses some of the inadequacies of the default-theory approach:

(1) the aforementioned unintuitive semantics based on consistency with a set of beliefs,

(2) the inability to reason about the defaults, e.g., given that "ravens are (normally) black" and "albino ravens are (normally) not black" we might want to conclude that "nonalbino ravens are normally black,"

(3) seemingly inconsistent sentences can coexist if one is an element of W and the other an element of D because the default rule would never be applied, e.g., "ravens are birds" and "normally ravens are not birds." The same applies if both sentences are elements of D: one of the default rules would not be applied.

AI approaches related to default reasoning are presented in [Fahlman 79] [Fahlman, Touretsky, van Roggen 81] [Touretsky 84] [Cottrell 84] [McCarthy 80] [McDermott and Doyle 80] [Moore 83].

The second approach to natural kinds representation in AI is prototype theory [Rosch 78]. According to this view natural kinds systems are organized as hierarchical taxonomies, or category systems. Attributes are seen at least in part as constructs of the perceiver. Membership in the extension of a kind or category is determined on the basis of similarity to a prototype [Quine 77]. One application of prototype theory has been the use of frames [Minsky 75]. Frames can be viewed as a network of nodes and relations, with the top levels representing things which are always true and lower levels consisting of terminals, or "slots," which may be filled with default assignments. Default assignments may later be displaced by new information which better fits the current instance or situation. Collections of related frames are linked together in *frame systems*. Such systems work well for prediction of properties given an individual, or for prediction of a kind given a list of properties.

Brachman [85] addresses the issue of whether default-oriented frames/semantic networks can represent arbitrary concepts and their interrelationships. It is argued that if any interesting nondefinitional property is potentially cancellable, then the interpretation of frames/nodes must be as strictly default

conditions thereby excluding definitions and even contingent universal statements. Reiter's default logic is capable of representing certain knowledge as uncancellable since a default theory includes a set W of first-order formulae. Yet the problems of semantics remain.

2. Conditional Logics

Conditional logics in general and N in particular are concerned with extensions of classical logics to include weaker forms of implication. System N [Delgrande 86] provides a propositional logic with the addition of a variably strict conditional operator $\Rightarrow$, where $A \Rightarrow B$ is interpreted as "all other things being equal, if A then B," or "ignoring exceptional conditions, if A then B," or "in the normal course of events, if A then B." Thus, $Raven(x) \Rightarrow Black(x)$ is interpreted as "in the normal course of events, ravens are black." Material conditionals exhibit the following properties:

(1) strengthening the antecedent: $(A \supset C) \vdash (A \wedge B) \supset C$

(2) modus ponens: $A, A \supset B \vdash B$

(3) transitivity: $((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

(4) law of the excluded middle: $(A \supset B) \vee (A \supset \neg B)$

(5) contraposition: $(A \supset B) \equiv (\neg B \supset \neg A)$

(6) strong deduction: $((A \wedge B) \supset C) \equiv (A \supset (B \supset C))$

Property (1), strengthening the antecedent, is also referred to as *left-downward monotonicity*. If the material conditional $A \supset B$ is true, then conditions can be conjoined to the antecedent, and the implication remains true. However, for representation of natural kinds we might want to assert a set of sentences such as:

$Raven(x) \Rightarrow Black(x)$

$Raven(x) \wedge Albino(x) \Rightarrow \neg Black(x)$

$Raven(x) \wedge Albino(x) \wedge Victim\_of\_oil\_spill(x) \Rightarrow Black(x)$

This property of strengthening the antecedent and possibly obtaining a change in the consequent to its negation is referred to as *left-downward nonmonotonicity*. The goal is to obtain a system in which a set of

such assertions is satisfiable while the antecedents are all true. We gain left-downward nonmonotonicity at the expense of modus ponens. This must be so because if we had:

$$A,\ B,\ A \Rightarrow C,\ A \wedge B \Rightarrow \neg C$$

then we could conclude both C and $\neg C$. We also lose strong deduction and contraposition in such systems.

Other properties of the material conditional we would like to see weakened for the representation of natural kinds are transitivity and the law of the excluded middle. For example, platypuses are mammals, mammals (normally) give birth to live young, but platypuses (normally) lay eggs (failure of transitivity). Also we might like to remain neutral about the relations between kinds and properties not usually associated with the kind (failure of the law of the excluded middle).

Similar systems have been developed for other types of reasoning, most notably those of Stalnaker [68] and Lewis [73] for the analysis of the traditional problems of counterfactual conditionals. Counterfactual conditionals are statements of the form "if something which is not the case had been the case, then something else would have been true." Conditionals concerning natural kinds, or *natural conditions*, and counterfactual conditionals share the characteristics of left-downward nonmonotonicity and failure of transitivity. They differ in that counterfactual reasoning deals with conditionals in which the antecedent is false while the conditional may be true or false. In the case of natural conditionals the antecedents are true, but the classical conditionals may be jointly inconsistent. Related work in analyzing counterfactual conditionals is dealt with in [Nute 75], subjunctive conditionals in [Pollock 76], and conditional obligation in [van Fraasen 72]. For discussions of the characterizations of conditional logics in general see [Chellas 75] [Nute 80] [Veltman 85] [van Benthem 84] [Jennings 87].

The general structure of the formal semantics for such systems is based on a set of worlds in which sentences of the language have "truth values." Each sentence has a definite value of "true" or "false" in each world. Thus three different meanings for "affirming" a proposition p can be distinguished:

(1) p is true in a single world (fact)

(2) p is true in a subset of worlds (hypothesis)

(3) p is true in all worlds in all models (valid).

Conditional statements are of interest when they are valid or used in hypotheses. Their use in hypotheses is generally as a "filter," i.e., to assume p as a hypothesis is to select the subset of worlds in which p is true.

The application of the possible worlds approach to the analysis of conditionals is due largely to Stalnaker [Pollock 76]. Stalnaker and Lewis view a counterfactual conditional as a "filter" which selects the "nearest" or "most similar" world i to the world in which the hypothesis is being tested, such that the antecedent is true at world i. In accordance with Lewis's interpretation, $A \Rightarrow B$, read "if it were the case that A then it would be the case that B, is true at a given world i if in the set of worlds (or "sphere") most similar to i that have A true, also have B true, and for no world in the sphere is $A \supset B$ false.

3. A Logic for Kinds

In the case of natural conditionals, the basis of the accessibility relation among possible worlds is uniformity, or "unexceptionalness." More formally, the accessibility relation E is said to hold between two worlds w_1 and w_2 (Ew_1w_2) just when w_2 is at least as unexceptional as w_1. $A \Rightarrow B$ is true with respect to a world w just when (1) the least exceptional worlds which have A true have B true also, and in no less exceptional world is $A \supset B$ false, or (2) A is false at all worlds accessible from w.

E is characterized by the following properties:

Reflexive: Eww for all worlds w.

Transitive: If Ew_1w_2 and Ew_2w_3 then Ew_1w_3.

Forward Connected: If Ew_1w_2 and Ew_1w_3 then either Ew_2w_3 or Ew_3w_2.

Forward connectedness ensures that every two worlds accessible from the same world are themselves comparable.

The language N consists of a denumerable set of atomic sentences P=$\{p_0, p_1, \cdots\}$, the logical connectives $\neg$ (negation) and $\supset$ (material conditional), and the variably strict conditional $\Rightarrow$. The connectives $\wedge$ (conjunction), $\vee$ (disjunction), and $\equiv$ (equivalence) can be defined in terms of negation and material implication.

Sentences of N are interpreted with respect to a model structure $M = <W, E, P>$ where W is a set, E is a reflexive, transitive, forward connected, and terminating binary relation on W, and P is a function which

maps atomic sentences onto subsets of W. W may be thought of as a set of possible worlds, E an accessibility relation on the worlds in W, and P a mapping of atomic sentences onto those worlds in W where a sentence is true. Ew_1w_2 is interpreted as "w_2 is at least as unexceptional as w_1." The symbol $\models^M_w A$ is used to indicate that A is a fact, i.e., is true in the model structure M in world w. More formally:

(i) $\models^M_w p_n$ iff $w \in P(p_n)$ for every n.

(ii) $\models^M_w \neg A$ iff not $\models^M_w A$.

(iii) $\models^M_w A \supset B$ iff if $\models^M_w A$ then $\models^M_w B$.

(iv) $\models^M_w A \Rightarrow B$ iff (a) there is a $w_1 \in W$ so that Eww_1 and $\models^M_{w_1} A$ and $\models^M_{w_1} B$ and there is no $w_2 \in W$ so that Ew_1w_2 and $\models^M_{w_2} A$ and $\models^M_{w_2} \neg B$, or (b) for every $w_1 \in W$ where Eww_1, $\models^M_{w_1} \neg A$.

If $\models^M_w A$, then A is true in model structure M in world w (a fact). If $\models A$, then A is *valid*, i.e., true in every world in every model structure. A is *satisfiable* iff $\neg A$ is not valid.

Since theorems of classical propositional calculus are theorems of N, any of the usual sets of axioms of PC and *modus ponens* may be adopted. In addition, the following formally characterizes the variable conditional operator $\Rightarrow$:

A0 $A \Rightarrow A$

A1 $((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow (B \wedge C))$

A2 $(A \Rightarrow B) \supset (((A \wedge B) \Rightarrow C) \supset (A \Rightarrow C))$

A3 $\neg (A \Rightarrow B) \supset ((A \Rightarrow C) \supset ((A \wedge \neg B) \Rightarrow C))$

A4 $((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset ((A \vee B) \Rightarrow C)$

A5 $((A \vee B) \Rightarrow C) \supset ((A \Rightarrow C) \vee (B \Rightarrow C))$

R0 If $B \supset C$ then infer $(A \Rightarrow B) \supset (A \Rightarrow C)$.

N is closed under the rules:

R1 If $A \equiv B$ then infer $(A \Rightarrow C) \equiv (B \Rightarrow C)$

R2 If $(B_1 \wedge \cdots \wedge B_n) \supset B$ then infer $((A \Rightarrow B_1) \wedge \cdots \wedge (A \Rightarrow B_n)) \supset (A \Rightarrow B)$ for every $n \geq 0$.

The desired result that $\{A_1 \Rightarrow B,\ A_1 \wedge A_2 \Rightarrow \neg B\}$ be satisfiable is obtained. Failure of transitivity is evidenced in the satisfiability of the following sets:

$\{A \Rightarrow B, B \Rightarrow C,\ \neg (A \Rightarrow C)\}$ and

$\{A \supset B, B \Rightarrow C,\ \neg (A \Rightarrow C)\}$.

We do, however, get the validity of:

$$(A \Rightarrow B) \supset ((B \supset C) \supset (A \Rightarrow C))$$

which is desirable in order to express sentences such as "ravens are (normally) black, black things are not white, so ravens are (normally) not white."

4. Thesis Overview

The approach to theorem-proving for N is presented in Chapter 2. The method of N-diagrams, a refutation procedure developed to test N-validity, incorporates the techniques of semantic tableaux. Section 1 of Chapter 2 provides an introduction to the method of N-diagrams. Section 2 provides background on tableau systems and semantic diagrams. The method of N-diagrams is then presented in detail in section 3. A summary of the approach is provided in section 4.

The proof of correctness of the theorem-proving algorithm is presented in Chapter 3. The correctness proof demonstrates that the method is consistent (section 1 of Chapter 3), and sound (section 2) and complete (section 3) with respect to the model theory.

Chapter 4 is concerned with the automation of the method of N-diagrams. Section 1 of Chapter 4 is an examination of the representation of a system of N-diagrams. The algorithm for the automated theorem prover is given in section 2 and the main data structures described in section 3. An analysis of the computational complexity of the algorithm is provided in section 4. Conclusions and directions for further research are presented in Chapter 5.

Chapter 2. A Theorem Prover for N

1. Introduction

The theorem prover for N is a refutation procedure based on the model theory of N rather than the axiomatic basis of N. An attempt is made to construct a falsifying model for a wff ω. If such a model can be constructed, ω is invalid. Otherwise a falsifying model is shown to be impossible, and ω is therefore valid.

The goal is to prove ω. A secondary goal is established to prove ¬ω satisfiable. If ¬ω is satisfiable then ω cannot be valid by the definition of truth in the model theory. If ¬ω is unsatisfiable then ω is valid. ¬ω is satisfiable if there exists a world w in some model structure M in which ¬ω is true.

Lower case Greek letters are used to denote formulas. A formula in which the main operator is ⇒, e.g., $A \Rightarrow B$, is referred to as a *γ-formula*. An *N-model* is a model structure $<W, E, P>$ in which the following conditions are met:

(1) For every world w_i, no subformula and its negation are both true at w_i.

(2) If $\models^M_w \gamma_1 \Rightarrow \gamma_2$ then (a) there exists a world w_i such that Eww_i and $\models^M_{w_i} \gamma_1$ and $\models^M_{w_i} \gamma_2$ and no w_j such that $\models^M_{w_j} \gamma_1$ and $\models^M_{w_j} \neg\gamma_2$, or (b) $\models^M_{w_i} \neg\gamma_1$ for all w_i such that Eww_i.

(3) If $\models^M_w \neg(\gamma_1 \Rightarrow \gamma_2)$ then there exists a world w_i such that Eww_i and $\models^M_{w_i} \gamma_1$ and $\models^M_{w_i} \neg\gamma_2$ and either (a) there exists no world w_j such that Ew_iw_j and $\models^M_{w_j} \gamma_1$ and $\models^M_{w_j} \gamma_2$, or (b) if there exists such a w_j, then Ew_jw_i.

(4) E relates the elements of W so that reflexivity, transitivity, and forward-connectedness hold.

Condition (1) says that the assignments of truth at worlds are consistent, i.e., true cannot be assigned to any subformula and its negation. Conditions (2) and (3) ensure that the truth conditions for the ⇒ operator are met at each world in the model.

The theorem-proving procedure attempts to construct an N-model for ¬ω in order to prove ω. If an N-model is found, ¬ω is satisfiable, and ω is invalid. The N-model for ¬ω serves as a counterexample. If, on the other hand, an N-model for ¬ω is impossible, then ¬ω is unsatisfiable, and ω is valid.

The proof procedure is an extension of the tableau-based semantic diagrams of Hughes and Cresswell [68] and Rescher and Urquhart [71]. All relevant reflexive, transitive, and forward-connected (RTFC) model configurations for $\neg\omega$ are generated and tested, where a *model configuration*, or *configuration*, is a diagrammatic representation of a model structure.[4] If a consistent configuration is found, the procedure returns the configuration. If every configuration is inconsistent, then ω is valid. Section 2 provides an introduction to tableau systems and semantic diagrams. The method of N-diagrams is then presented in section 3.

2. Background

2.1. Tableau Systems

The procedure for testing N-validity is a tableau-based system. The roots of such systems can be traced back to Gentzen [35]. His *sequent calculus* is generally applied in an "upsidedown" fashion, starting with the desired result and working upward to axioms. Beth [59] and Hintikka [55] developed proof procedures called tableau systems which formulate the problem "upsidedown" from the beginning. The procedures are refutation systems, i.e., an attempt is made to refute a given formula. If the attempt fails, the formula should be valid.

Smullyan [68] applied these ideas to classical logic, simplifying the methods and making them more elegant. Oppacher and Suen [86] implemented an automated tableau-based theorem prover for full first-order logic based on Smullyan's analytic tableau in which the control structure has been augmented with heuristics. Extensions of Beth's intuitionistic tableau systems were developed by Fitting [83]. Tableau systems have also evolved for modal logics [Kripke 63, Hughes and Cresswell 68, Fitting 83] and temporal logics [Rescher and Urquhart 71].

---

[4] By *relevant* model configuration is meant a model configuration in which only relevant value assignments at each world are considered. Every atomic sentence has a definite truth value at each world in a model structure. For example, there are many models in which $\models^{M}_{w} \gamma_1 \Rightarrow \gamma_2$ and $\models^{M}_{w_1} \gamma_1$ and $\models^{M}_{w_1} \gamma_2$ and Eww_1. These models vary in the assignment of truth values to other atomic sentences, but that is of no concern. The main concern is that γ_1 and γ_2 are both true at w_1 and w_1 is accessible to w thereby confirming $\models^{M}_{w} \gamma_1 \Rightarrow \gamma_2$.

The method of tableau involves an attempt to construct a falsifying model for a given formula. If a falsifying model can be constructed, then the formula is invalid. Otherwise it is shown that such a model is impossible to construct, and the formula is therefore valid. The standard notion of propositional formula is characterized in the following.

Definition. The set of *formulas* is precisely defined by:

(1) Any propositional variable (atomic formula) is a formula.

(2) If X is a formula, then so is $\neg X$.

(3) If X, Y are formulas, then so are $(X \wedge Y)$, $(X \vee Y)$, $(X \supset Y)$.

Definition. According to the *uniqueness of decomposition* property, for every formula X, one and only one of the following conditions holds [Smullyan 68; proofs that propositional languages have this property are given in Church 56, Kleene 52]:

(1) X is a propositional variable.

(2) There is a unique formula Y such that $X = \neg Y$.

(3) There is a unique pair X_1, X_2 and a unique binary connective b such that $X = (X_1\ b\ X_2)$.

Definition. The *degree of a formula* refers to the number of occurrences of the logical connectives $\wedge$, $\vee$, $\supset$, $\neg$ within a formula:

(1) A variable is of degree 0.

(2) If X is of degree n, then $\neg X$ is of degree $n + 1$.

(3) If X, Y are of degree n_1, n_2, then $X \wedge Y$, $X \vee Y$, $X \supset Y$ are each of degree $n_1 + n_2 + 1$.

Definition. The notion of *subformula* as described in [Smullyan 68] follows. An *immediate subformula* is given by:

(1) Propositional variables have no immediate subformulas.

(2) $\neg X$ has X as an immediate subformula and no others.

(3) Formulas $X \wedge Y$, $X \vee Y$, $X \supset Y$ have X, Y as immediate subformulas and no others.

A *subformula* is given by:

(1) If X is an immediate subformula of Y, or if X is identical with Y, then X is a subformula of Y.

(2) If X is a subformula of Y, and Y is a subformula of Z, then X is a subformula of Z.

Then any subformula of X must have degree less than the degree of X.

Definition. A *signed* formula refers to an unsigned formula X preceded by T or F. Informally, TX asserts that formula X is true, FX asserts that X is false.

Definition. A *tableau system* consists of (1) a tree with root FZ, where Z is the formula to be proven, and (2) rules for extending branches of the tree. For the propositional case there are two types of rules, α-rules and β-rules, for the assignment of truth values to the immediate subformula of a formula. These rules are given in Table 2.1.[5]

Table 2.1. Summary of α-rules and β-rules.

α	α_1	α_2	β	β_1	β_2
$T(X \wedge Y)$	TX	TY	$F(X \wedge Y)$	FX	FY
$F(X \vee Y)$	FX	FY	$T(X \vee Y)$	TX	TY
$F(X \supset Y)$	TX	FY	$T(X \supset Y)$	FX	TY
$F(\neg X)$	TX	TX	$T(\neg X)$	FX	FX

Note that α_1, α_2 are *direct consequences* of α while at least one of β_1, β_2 follows from β. In terms of Tarskian semantics, $\models_w^M \alpha$ iff $\models_w^M \alpha_1$ and $\models_w^M \alpha_2$, and $\models_w^M \beta$ iff $\models_w^M \beta_1$ or $\models_w^M \beta_2$. The rules are further summarized in the table below, where "|" means or:

[5] The classification of $F\neg X$ as an α-formula and $T\neg X$ as a β-formula is arbitrary. It is, however, desirable to categorize them as separate formula types if characterizations of the rules are to be examined. For example, consider some properties of conjugation. The conjugate of a signed formula is the result obtained from changing T to F or F to T. Thus, the conjugate of any α is a β, and the conjugate of a β is an α. These properties would not hold if $F\neg X$ and $T\neg X$ were both α- or β-formulas.

α	β
α_1 α_2	β_1 | β_2

For modal and first-order augmentation of the branch extension rules see Appendix 2.

Rules are applied in the following way. Given an α-formula on branch θ of tree T, extend branch θ and each of its branches with α_1, α_2. Given a β-formula on branch θ, extend θ and each of its branches by branching β_1 to the left and β_2 to the right. A branch is *closed* if:

(1) For some formula *A*, *TA* and *FA* both appear on a branch θ,

(2) $T\perp$ appears on θ, where $\perp$ is the logical constant *false*, or

(3) $F\top$ appears on θ, where $\top$ is the logical constant *true*.

A branch is open otherwise. A tree or tableau is *closed* if all of its branches are closed. A closed tableau for *FZ* is a proof of *Z* [Smullyan 68]. Once a rule is applied to an α or β, that formula is marked and never used again, not even in deriving a contradiction to close a branch.

The following example illustrates the tableau method. The problem is to prove that $(((A \supset C) \wedge (B \supset C)) \wedge (A \vee B)) \supset C$ is a tautology. The parenthesized numbers to the left of a line are used only for identification. The parenthesized numbers to the right indicate the line from which the current line was derived. The proof is given in Figure 2.1.

The α-rule "given $F(X \supset Y)$ infer TX, FY" is applied to line (1) yielding (2) and (3). A signed variable (line (3)) cannot be used to create new lines, but (2) yields (4) and (5), and (4) yields (6) and (7). Lines (5), (6), and (7) are β-formulas. Line (5) requires a branching to (8) and (9). The branch extension rules extend each path which goes through the α- or β-formula. Thus, line (6) requires branching from (8) to (10) and (11) as well as from (9) to (12) and (13). Lines (8) and (10) contain a contradiction, *TA* and *FA*, so that branch is closed (marked with an *X*), and so on. All branches are closed, so the tableau is closed and $(((A \supset C) \wedge (B \supset C)) \wedge (A \vee B)) \supset C$ is a tautology.

After a finite number of steps a tableau is *complete* since:

(1) $F((((A \supset C) \wedge (B \supset C)) \wedge (A \vee B)) \supset C)$

(2) $T(((A \supset C) \wedge (B \supset C)) \wedge (A \vee B))$ (1)

(3) FC (1)

(4) $T((A \supset C) \wedge (B \supset C))$ (2)

(5) $T(A \vee B)$ (2)

(6) $T(A \supset C)$ (4)

(7) $T(B \supset C)$ (4)

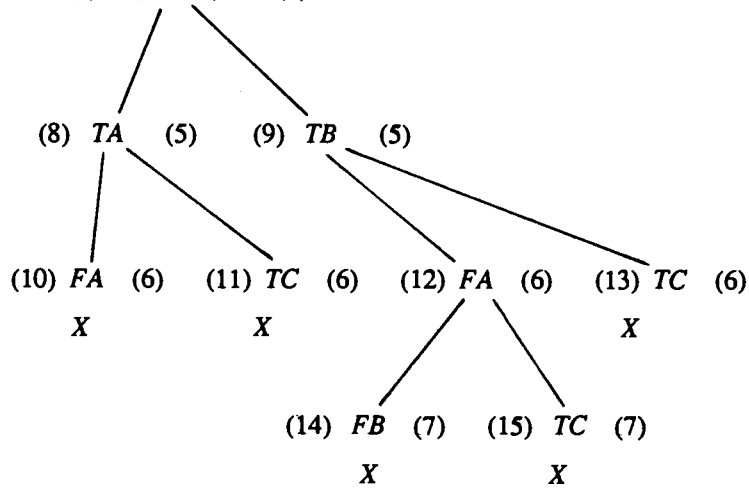


Figure 2.1. A tableau proof of $((A \supset C) \wedge (B \supset C) \wedge (A \vee B)) \supset C$.

(1) Application of the branch extension rules decomposes formulas into immediate subformulas.

(2) An immediate subformula of X must have degree less than X.

(3) Signed variables have no immediate subformulas.

(4) Each line in a tableau is used only once to create new lines (except for signed variables).

Smullyan [68] suggests some heuristics for tableau construction:

(1) Close a branch as soon as a contradiction is reached.

(2) Give priority to α-rules. Repetition of α_1, α_2 to all subsequent branches is then avoided. α_1, α_2 occur instead with α above the branching.

(3) Mark a line once it has been used to indicate it need not be considered again.

The method of tableau can also be used to show that $X_1, X_2, ..., X_n \vdash Y$, i.e., Y is a truth functional consequence of $X_1, X_2, ..., X_n$. A tableau can be constructed starting with $F((X_1 \wedge X_2 \wedge \cdots \wedge X_n) \supset Y)$ or starting with:

TX_1

TX_2

:

:

TX_n

FY.

2.2. Semantic Diagrams

Since the truth conditions for the variable conditional operator $\Rightarrow$ are based on a possible-worlds semantics, a proof procedure for N must have the capacity to manipulate worlds. This capacity is evidenced in the method of semantic diagrams of Hughes and Cresswell [68] for modal logics and Rescher and Urquhart [71] for temporal logics. Let us examine the semantic diagrams of Hughes and Cresswell in more detail. The method is essentially a tableau technique consisting of:

(1) *rectangles* representing the relevant state of affairs or conditions which hold at worlds,

(2) *labels* identifying rectangles (worlds),

(3) *arrows* indicating accessibility, i.e., an arrow from rectangle w_i to rectangle w_j indicates that w_j is accessible from w_i, and

(4) *rules* for building the diagrams.

An attempt is made to construct a falsifying model for a given formula utilizing the method. Let us first review the propositional case which obviates the accessible worlds machinery. This machinery will later be examined in detail with a modal example.

Consider the propositional formula $\omega = (A \wedge B) \supset (A \vee B)$. The goal is to provide a model in which ω is shown to be false or to prove such a model impossible to construct. The notion of Boolean valuations is

used instead of interpretations in the context of semantic diagrams. A valuation assigns 1 (true) or 0 (false) to formulas and subformulas. The truth value for ω in a falsifying model is 0. Thus, we begin by placing a zero under the main operator of ω:

w_1	$(A \wedge B) \supset (A \vee B)$
	0

The application of the rules of propositional logic forces the assignment of the value 1 to the antecedent and 0 to the consequent of a false implication (an α-rule):

w_1	$(A \wedge B) \supset (A \vee B)$
	1 0 0

Application of the rules for a true ∧ and a false ∨ (α-rules) result in the following assignments:

w_1	$(A \wedge B) \supset (A \vee B)$
	1 1 1 0 0 0 0

Inconsistencies arise in the value assignment of both 0 and 1 to *A* and to *B*. The rectangle containing such an inconsistency is said to be inconsistent. Thus a falsifying model is impossible to construct, and ω is valid. The α- and β-rules are presented in Table 2.2 in the context of semantic diagrams.

Efficiency is improved if α-rules are applied first, then β-rules. The α-rules are applied by assigning the appropriate value(s) to the operand(s) of an α-formula. For example, given $A \wedge B$ assigned 1, *A* is assigned 1, and *B* is assigned 1. β-formulas are dealt with in the following way. According to Hughes and Cresswell's system a † is placed under the leftmost β in a rectangle w_j. Limiting the number of †'s to one per rectangle ensures that at most three alternatives are generated from any one rectangle w_j. The rule for alternatives (β-rules) creates two (or three) new rectangles, $w_{j(i)}$, $w_{j(ii)}$ (or $w_{j(i)}$, $w_{j(ii)}$, $w_{j(iii)}$) so that each is a replica of w_j to which distinct alternative-case value assignments have been made, e.g., Figure 2.2. Rectangle w_j is inconsistent iff each alternative case $w_{j(i)}$, $w_{j(ii)}$, $w_{j(iii)}$ is inconsistent.

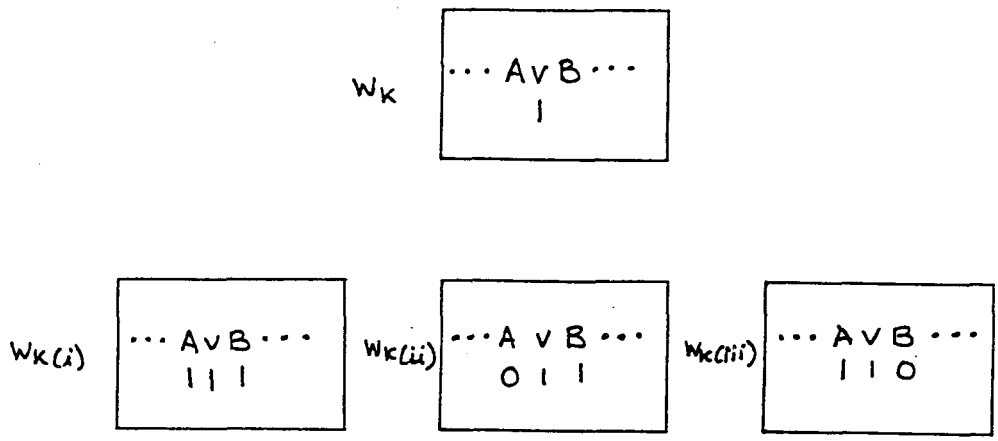


Figure 2.2. Alternative rectangles.

Table 2.2. Summary of α-rules and β-rules for value assignments.

α	α_1	α_2	β	Alternative cases		
A ∧ B	A	B	A ∨ B	A B	A B	A B
1	1	1	1	1 1	0 1	1 0
A ∨ B	A	B	A ∧ B	A B	A B	A B
0	0	0	0	1 0	0 1	0 0
A ⊃ B	A	B	A ⊃ B	A B	A B	A B
0	1	0	1	1 1	0 1	0 0
¬ A	A		A ≡ B	A B	A B	
0	1		1	1 1	0 0	
¬ A	A		A ≡ B	A B	A B	
1	0		0	1 0	0 1	

The rule for alternatives is illustrated with $\omega = ((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$. We begin by writing ω in a rectangle labelled w_1 and placing a zero under the main operator:

w_1 | $((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$
 0

The application of the rules of propositional logic forces the assignment of the value 1 to the antecedent and 0 to the consequent of a false implication (an α-rule):

w_1 | $((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$
 1 0 0

These assignments in turn require that the following value assignments be made:

w_1 | $((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$
 1 1 1 0 1 0 0

A cross is placed under the leftmost β:

w_1

$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

1 1 1 0 1 0 0

†

New rectangles for the alternative cases are created:

w_1 (i)

$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

1 1 1 1 1 0 1 0 0

†

w_1 (ii)

$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

0 1 1 1 1 0 1 0 0

†

w_1 (iii)

$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

0 1 0 1 1 0 1 0 0

†

New rectangles for the alternative cases to $w_{1(i)}$ are created:

w_1 (i)(i)

$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

1 1 1 1 1 1 1 0 1 0 0

w_1 (i)(ii)

$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

1 1 1 1 0 1 1 0 1 0 0

w_1 (i)(iii)

$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

1 1 1 1 0 1 0 0 1 0 0

Each alternative case leads to an inconsistency, so rectangle $w_{1(i)}$ is inconsistent. New rectangles are next created for the alternative cases to $w_{1(ii)}$:

w_1 (ii)(i)	$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$ 0 1 1 1 1 1 1 0 1 0 0
w_1 (ii)(ii)	$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$ 0 1 1 1 0 1 1 0 1 0 0
w_1 (ii)(iii)	$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$ 0 1 1 1 0 1 0 0 1 0 0

Each alternative case leads to an inconsistency, so rectangle $w_{1(ii)}$ is inconsistent. Finally, new rectangles for the alternative cases to $w_{1(iii)}$ are created:

w_1 (iii)(i)	$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$ 0 1 0 1 1 1 1 0 1 0 0
w_1 (iii)(ii)	$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$ 0 1 0 1 0 1 1 0 1 0 0
w_1 (iii)(iii)	$((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$ 0 1 0 1 0 1 0 0 1 0 0

Each alternative case leads to an inconsistency, so $w_{1(iii)}$ is inconsistent. Each alternative case to w_1, i.e., $w_{1(i)}$, $w_{1(ii)}$, $w_{1(iii)}$, is inconsistent, so w_1 is inconsistent, and ω is valid.

Next examine the propositional formula $\omega = (A \supset C) \supset (C \supset A)$. A zero is placed under the main operator and the following forced value assignments made:

w_1

(A ⊃ C) ⊃ (C ⊃ A)
1 0 1 0 0
†

New rectangles are created for the alternative cases to w_1:

w_1 (i)

(A ⊃ C) ⊃ (C ⊃ A)
1 1 1 0 1 0 0

w_1 (ii)

(A ⊃ C) ⊃ (C ⊃ A)
0 1 1 0 1 0 0

w_1 (iii)

(A ⊃ C) ⊃ (C ⊃ A)
0 1 0 0 1 0 0

Rectangles $w_{1(i)}$ and $w_{1(iii)}$ lead to inconsistencies, but $w_{1(ii)}$ is consistent. The value assignment of 0 to A and 1 to C demonstrates a falsifying model, and ω is invalid.

This method is just a modified tableau system. The first step, which places a zero under the main operator, corresponds to a tableau which starts with the signed formula $F\omega$, where ω is the given formula. Then the rules of propositional logic (α- and β-rules) are applied in both systems until no more rules apply. For example, the proof of validity of $\omega = (((A \supset B) \wedge (B \supset C)) \supset (A \supset C))$ using Smullyan's method is as follows:

(1) $F(((A \supset B) \wedge (B \supset C)) \supset (A \supset C))$

(2) $T((A \supset B) \wedge (B \supset C))$ (1)

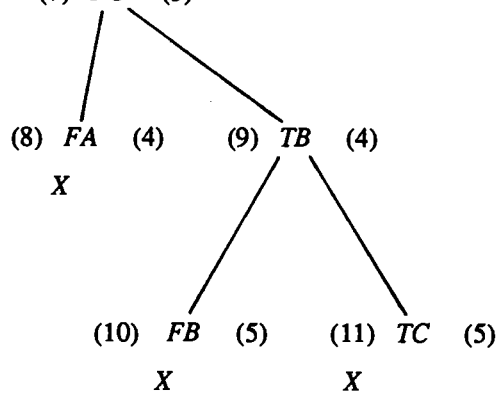
(3) $F(A \supset C)$ (1)

(4) $T(A \supset B)$ (2)

(5) $T(B \supset C)$ (2)

(6) TA (3)

(7) FC (3)



All branches are closed, so the formula is valid. Next examine the tableau for $\omega = ((A \supset C) \supset (C \supset A))$ using Smullyan's method:

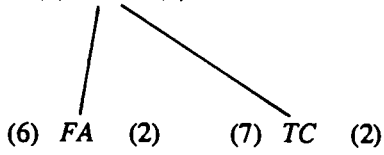
(1) $F((A \supset C) \supset (C \supset A))$

(2) $T(A \supset C)$ (1)

(3) $F(C \supset A)$ (1)

(4) TC (3)

(5) FA (3)



There is at least one open path, so $(A \supset C) \supset (C \supset A)$ is invalid.

Next consider a modal example. Let L be the necessity operator and M the possibility operator. Asterisks are used to mark the occurence of a modal operator within a rectangle. The rule for putting in asterisks requires that:

(1) an asterisk be placed above each L assigned 1 and each M assigned 0, and

(2) an asterisk be placed under each L assigned 0 and each M assigned 1.

Rule (1) marks ν-formulas, formulas in which the main operator is an L assigned 1 or an M assigned 0. ν-formulas $L\omega$ ($M\omega$) force the assignment of 1 (0) to ω at *all* accessible worlds. Rule (2) marks π-formulas, formulas in which the main operator is an L assigned 0 or an M assigned 1. π-formulas $L\omega$ ($M\omega$) force the assignment of 0 (1) to ω at *some* accessible world.

There are four rules for a new world. Rules (1) and (2) below are ν-rules and (3) and (4) are π-rules:

(1) If an asterisk occurs above $L\omega$ in world w_i, then in all worlds accessible from w_i, ω must be assigned 1.

(2) If an asterisk occurs above $M\omega$ in world w_i, then in all worlds accessible from w_i, ω must be assigned 0.

(3) If an asterisk occurs beneath $L\omega$ in world w_i, then there must exist a world accessible from w_i in which ω is assigned 0.

(4) If an asterisk occurs beneath $M\omega$ in world w_i, then there must exist a world accessible from w_i in which ω is assigned 1.

The method is illustrated by testing the T-validity of $\omega = Mp \equiv \neg L \neg p$. T is a modal logic characterized by a reflexive accessibility relation. Wff ω is written in a rectangle labelled w_1, a zero placed under the main operator, and the false equivalence marked with a cross as in Figure 2.3.

The rule for alternatives is applied creating $w_{1(i)}$ and $w_{1(ii)}$ as in Figure 2.4. $w_{1(i)}$ contains the case in which Mp is assigned 1 and $\neg L \neg p$ is assigned 0. This forces the assigment of 1 to $L\neg p$, 1 to $\neg p$, and 0 to p. The L assigned 1 is marked with an asterisk above, and the M assigned 1 is marked with an asterisk below. Applying the π-rule for a true possibility operator, there must exist some world w_2 in which p is true. However, the asterisk above L requires that in all worlds accessible from $w_{1(i)}$ $\neg p$ must be true. Rectangle w_2 is inconsistent since p is assigned both 0 and 1, and $w_{1(i)}$ is therefore inconsistent. Similarly $w_{1(ii)}$ leads to an inconsistency. Therefore, w_1 is inconsistent and ω is valid.

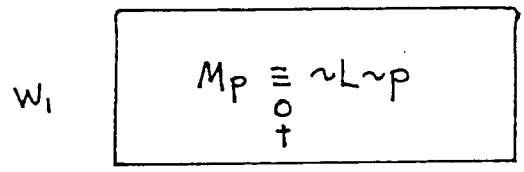


Figure 2.3. Initial rectangle for $\omega = Mp \equiv \neg L \neg p$.

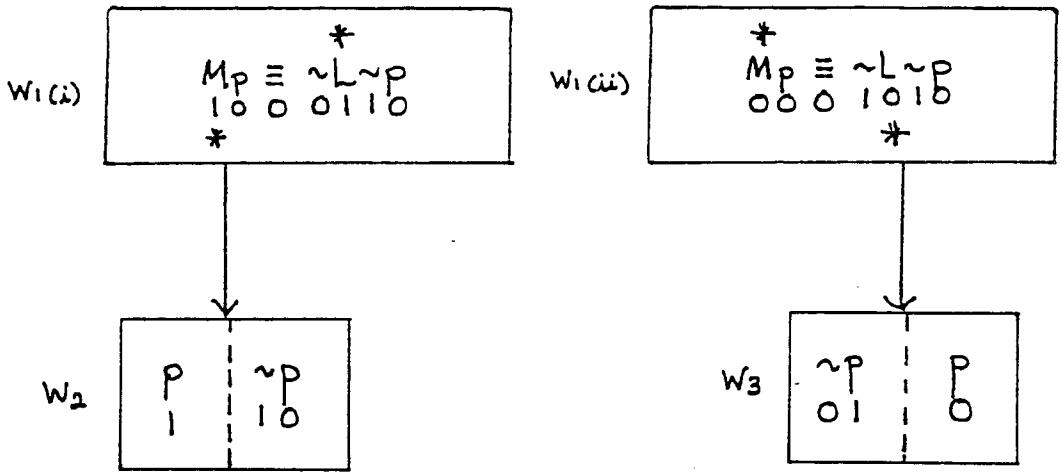


Figure 2.4. Complete system for $\omega = Mp \equiv \neg L \neg p$.

To summarize the method of semantic diagrams, we begin by writing the formula to be tested in a rectangle labelled w_1 and placing a 0 under the main operator. In each rectangle w_i rules are applied first which infer direct consequences (α- and ν-rules). Rectangles other than w_1 are built by applying rules:

(1) rule for putting in asterisks,

(2) rule for a new world,

(3) rule for alternatives.

The system is *complete* for a wff ω when the rules have been applied as often as possible. A rectangle w_i in a complete system is inconsistent if:

(1) both 0 and 1 have been assigned to the same subformula in w_i,

(2) an inconsistent rectangle is accessible from w_i, or

(3) all of its alternatives are inconsistent.

A formula ω is valid iff the first rectangle, w_1, in a complete system of semantic diagrams is inconsistent.

3. N-Diagrams

3.1. The Method

The theorem-prover for N is a refutation procedure based on the model theory of N as opposed to the axiomatic basis of N. In order to prove ω, an attempt is made to construct a consistent N-model for $\neg\omega$. An extension of the semantic diagrams technique for modal and temporal logics is used to build a structure from which all relevant N-models can be generated. If any one of these N-models is consistent, then $\neg\omega$ is satisfiable and ω invalid. The consistent N-model serves as a counterexample. If, on the other hand, every N-model is inconsistent, then $\neg\omega$ is unsatisfiable, and ω is therefore valid.

The theorem prover is comprised of two parts. The first part builds a structure which characterizes possible N-models for $\neg\omega$. The second part generates and tests diagrammatic representations of N-models for $\neg\omega$ called *configurations*. In the following description of N-diagrams refer to Figure 2.5 which illustrates the truth conditions for the variably strict conditional operator $\Rightarrow$.

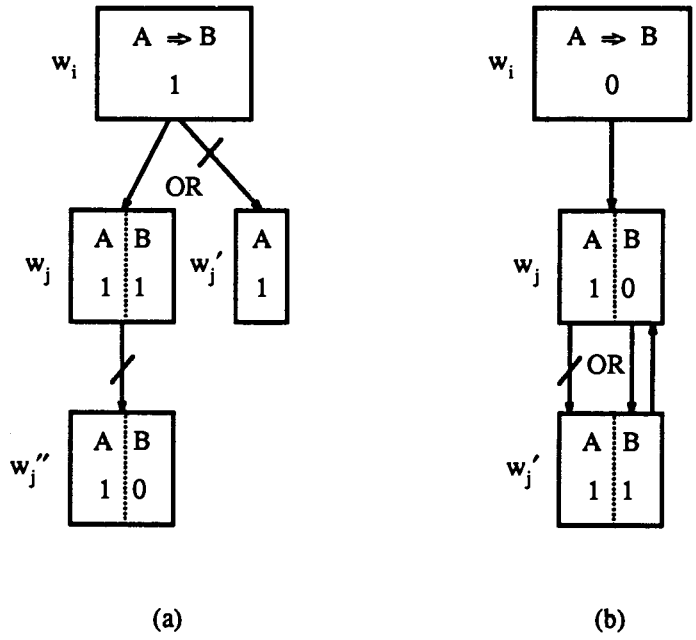


Figure 2.5. N-diagrams representing the truth conditions for the ⇒ operator. Figure 2.5(a) illustrates the conditions under which $A \Rightarrow B$ is true at a world w_i. Figure 2.5(b) shows the conditions under which $A \Rightarrow B$ is false at w_i.

A system of N-diagrams consists of:

(1) rectangles,

(2) labels,

(3) arrows,

(4) not-arrows,

(5) ORs, and

(6) rules for constructing the diagrams.

There are two types of rectangles: (a) rectangles representing the relevant state of affairs or conditions which hold at a world, and (b) rectangles representing the conditions or constraints on accessibility from a world. Associated with each rectangle is a label, an element from the set W of an N-model. A rectangle of type (a) labelled w_i contains the set of relevant sentences of N which are true at w_i. For example, the rectangle labelled w_i in Figure 2.5(a) represents a world w_i in which $A \Rightarrow B$ is true, whereas Figure 2.5(b)

represents a world w_i in which $A \Rightarrow B$ is false. Associated with a rectangle of type (b) is a primed label, e.g., rectangle w_j'' of Figure 2.5(a) contains the conditions which cannot hold at any rectangle w_k such that there exists an arrow from w_j to w_k.

An arrow is used to represent the accessibility relation E. An arrow from rectangle w_i to rectangle w_j indicates that Ew_iw_j holds, i.e., the world represented by rectangle w_j is accessible from the world represented by rectangle w_i. The symbolism Aw_iw_j is used to indicate that there exists an arrow in an N-diagram from rectangle w_i to rectangle w_j. Thus an arrow Aw_iw_j corresponds to Ew_iw_j in an N-model. So world w_j is accessible from w_i in Figure 2.5(b) since there exists an arrow from rectangle w_i to rectangle w_j. There is an implicit arrow from each rectangle to itself because the accessibility relation is reflexive.

A not-arrow, $\not\rightarrow$, leaving rectangle w_i indicates that there are constraints imposed upon any rectangle w_j such that Aw_iw_j exists. A rectangle into which a not-arrow enters contains the prohibited conditions. The conditions are forbidden by the definition of truth in an N-model. For example, Figure 2.5(a) represents a situation in which there can be no arrow Aw_jw_j'' where A is true and B false at rectangle w_j''.

An OR in a diagram indicates that either one accessibility path or the other must hold. There are two cases in which ORs occur:

(1) between an arrow from w_i to w_j and a not-arrow from w_i, where rectangle w_j was created due to a true $\Rightarrow$-subformula in w_i,

(2) between a not-arrow from w_j to w_j' and a pair of arrows, one from w_j to w_j' and the *return-arrow* from w_j' back to w_j.

Case (1) arises when $\gamma_1 \Rightarrow \gamma_2$ is assigned 1 at w_i. Then there must be an accessible world w_j where γ_1 and γ_2 are both true, or γ_1 is false at all accessible worlds. Case (2) arises when $\gamma_1 \Rightarrow \gamma_2$ is assigned 0 at w_i. Then there must be an accessible world w_j where γ_1 is true and γ_2 false either there is no world w_j' where γ_1 and γ_2 are both true, *or* $Aw_j'w_j$ coexists. An example of case (1) is in Figure 2.5(a) where either:

(a) there is an arrow Aw_iw_j and a rectangle w_j where A and B are both true, and no arrow Aw_jw_k such that A is true and B false at rectangle w_k, OR

(b) for every arrow Aw_iw_j, A is false at rectangle w_j.

Case (2) is illustrated in Figure 2.5(b) where either:

(a) there exists no arrow Aw_jw_j' where A and B are both true at rectangle w_j', OR

(b) if there is such an arrow Aw_jw_j', then $Aw_j'w_j$ must also exist.

Since N subsumes propositional logic, the valid wffs of classical propositional logic are valid wffs of N. The theorem-prover for N, then, requires the propositional rules for the assignment of truth values to the immediate subformulas of propositional formulas plus a rule for the assignment of truth values to the immediate subformulas of γ-formulas. Thus the method of N-diagrams consists of three types of rules for the assignment of truth values: α-rules, β-rules, and γ-rules. Truth assignments are made in terms of Boolean valuations rather than interpretations. The α-rules and β-rules are summarized in Table 2.2. Apply α-rules by assigning the appropriate value(s) to the operand(s) of an α-subformula. For example, given the α-formula $A \supset B$ assigned 0, assign 1 to A and 0 to B. Assignments to operand(s) made by the application of α-rules follow as *direct consequences* from the value assigned the α-subformula.

The *rule for crosses* serves to mark a β-subformula. A cross is placed under the leftmost β-subformula in rectangle w_j. Apply β-rules in the construction of N-diagrams according to the following rule:

Rule for alternatives. If a † occurs beneath an equivalence (or other β-subformula)in rectangle w_j, draw two (three) new rectangles $w_{j(i)}$, $w_{j(ii)}$ ($w_{j(i)}$, $w_{j(ii)}$, $w_{j(iii)}$) which are identical copies of w_j. Assign a distinct alternative-case value assignment to each new rectangle.

Note that no arrows are drawn from rectangle w_j to its alternatives. The arrows entering w_j are copied over to each alternative, however. We now have, in effect, a system of two (three) N-diagrams, identical to the original except that one of $w_{j(i)}$, $w_{j(ii)}$ ($w_{j(i)}$, $w_{j(ii)}$, $w_{j(iii)}$) replaces w_j in each.

A formula or subformula whose main operator is a $\Rightarrow$ is referred to as a γ-formula or γ-subformula. The *rule for asterisks* serves to mark a γ-subformula. An asterisk is placed under each $\Rightarrow$ assigned 0 or 1. The γ-rules are stated as follows:

(1) $\models^M_w A \Rightarrow B$ iff (a) there exists a $w_1 \in W$ such that Eww_1 and $\models^M_{w_1} A$ and $\models^M_{w_1} B$ and there exists no $w_2 \in W$ such that Ew_1w_2 and $\models^M_{w_2} A$ and $\models^M_{w_2} \neg B$, or (b) for all $w_1 \in W$ such that Eww_1, $\models^M_{w_1} \neg A$.

(2) $\models^M_w \neg(A \Rightarrow B)$ iff there exists a $w_1 \in W$ such that Eww_1 and $\models^M_{w_1} A$ and $\models^M_{w_1} \neg B$ and either (a) there exists no $w_2 \in W$ such that Ew_1w_2 and $\models^M_{w_2} A$ and $\models^M_{w_2} B$, or (b) if there is such a w_2 then Ew_2w_1.

The γ-rules are expressed in terms of N-diagram construction in the *rules for new worlds*:

(1) If an asterisk occurs beneath a $\Rightarrow$ assigned 1 at w_i, create a new rectangle w_j in which the antecedent is true and the consequent is true. (Refer to Figure 2.5(a).) Place an arrow from rectangle w_i to new rectangle w_j. Create another rectangle w_j' in which the antecedent is true. Place a not-arrow from rectangle w_i to rectangle w_j'. Place an OR between the arrow from w_i to w_j and the not-arrow from w_i to w_j'. Create another rectangle w_j'' in which the antecedent is true and the consequent false. Place a not-arrow from rectangle w_j to rectangle w_j''.

(2) If an asterisk occurs beneath a $\Rightarrow$ assigned 0 at w_i, create a new rectangle w_j in which the antecedent is true and the consequent false. (Refer to Figure 2.5(b).) Place an arrow from rectangle w_i to rectangle w_j. Create another rectangle w_j' in which the antecedent and the consequent are both true. Place a not-arrow from rectangle w_j to rectangle w_j'. Place an arrow from rectangle w_j to rectangle w_j' and from rectangle w_j' to rectangle w_j. Place an OR between the not-arrow and the double-arrows connecting w_j and w_j'.[6]

The method of N-diagrams thus provides tools for model construction:

(1) labelled rectangles representing worlds in an N-model,

(2) a set of sentences of N within a rectangle w_i denoting the set of relevant true sentences at world w_i,

(3) arrows representing the accessibility relation E between worlds,

(4) unlabelled rectangles containing sentences of N which denote constraints on accessibility from a world,

---

[6] In practice the primed labels are dropped from rectangles representing the conditions on accessibility from a rectangle. They have been used here to avoid confusion.

(5) not-arrows which emanate from labelled rectangles, w_i, and enter unlabelled rectangles denoting constraints on accessibility from w_i,

(6) ORs denoting alternative states of affairs either of which satisfy the definition of truth for $\Rightarrow$, and

(7) rules for assigning consequential truth values to subformula given the initial assignment of 0 to ω, e.g., α-, β-, γ-rules, rules for crosses, alternatives, asterisks, and new worlds.

The theorem-prover for N (NTP) is based upon this method of N-diagrams. The application of the method of N-diagrams to the theorem prover is developed with a simple, but inefficient NTP algorithm called NTP1. A more efficient algorithm is presented in a later section. Algorithm NTP1 is given below:

Algorithm NTP1

Input: wff

Output: valid, invalid

1. Initialize the system of N-diagrams
   - 1.1. Write the wff inside a rectangle
   - 1.2. Label the rectangle w_1
   - 1.3. Assign 0 to the main operator of the wff
2. Build "characterizing structure" (or semi-complete structure)
   - 2.1 Apply rules of the method as often as possible
3. Generate RTFC configurations
4. Test RTFC configurations
5. If a consistent RTFC configuration is found then return the configuration
   Else wff is VALID

To prove the given wff ω we try to find a configuration in which $\neg\omega$ is satisfiable. Steps 1 and 2 generate a structure which characterizes N-models for $\neg\omega$. This "characterizing structure" is not an N-model because it contains ORs. Removal of the ORs from the characterizing structure generates alternative structures called *templates*. These templates are not yet N-models because they are not forward-connected. Thus sets of arrows are added to the templates and tested for consistency (steps 3 and 4). A template to which a set of arrows has been added is called a *configuration*. If a consistent RTFC configuration for $\neg\omega$ is found, ω is invalid. Otherwise ω is valid. The next sections examine the steps of the algorithm in detail and provide examples.

3.2. Building the Semi-Complete Structure

The approach to theorem-proving presented here is one of model construction. The goal is to find a consistent model for $\neg\,\omega$. Thus the characteristics shared by all possible relevant consistent models must be determined. What worlds are required for such a model? What sentences must be true at each world in the model? How must these worlds be related?

The method of N-diagrams is employed to answer these questions. The procedure is illustrated with an example. Let $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$. Write ω into a rectangle labelled w_1 and place a 0 under the main operator:

w_1	$(((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$
	0

Apply the propositional α- and β-rules to obtain consequential value assignments to subformula of ω:

w_1	$(((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$
	1 1 1 0 0

Apply the rule for asterisks to mark each $\Rightarrow$-subformula:

w_1	$(((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$
	1 1 1 0 0
	* * *

Apply the rules for new worlds wherever an asterisk occurs. We obtain the diagram of Figure 2.6. The structure obtained once the rules have been applied as often as possible is referred to as a *semi-complete system of N-diagrams*, or *semi-complete structure* or SCS for short. It is semi-complete in the sense that more accessibility relations must be added since forward-connectedness and transitivity do not hold in such structures consisting of more than two worlds. The semi-complete structure for $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$ is shown in Figure 2.6. Such a structure is the characterizing structure referred to earlier.

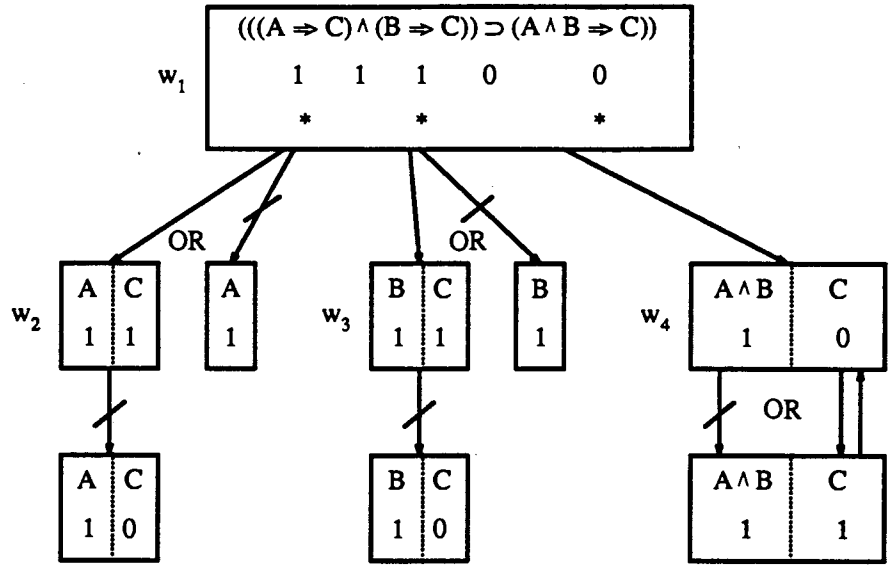


Figure 2.6. Semi-complete system of N-diagrams for $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$.

Note that for ω only one semi-complete structure exists. Many are possible, though, if a wff contains a β-subformula. In the case that a $\Rightarrow$-subformula occurs within the scope of a β-subformula, for example, we get quite different semi-complete structures. Consider the SCS for $\omega = \neg((A \Rightarrow B) \vee (B \Rightarrow A))$ shown in Figure 2.7. Because ω is a β-formula with $\Rightarrow$-subformulas $A \Rightarrow B$, $B \Rightarrow A$, we obtain a system of three SCS, (a), (b), (c), each of which is used to generate configurations.

The SCS specifies a limit on the worlds required in any consistent relevant model and specifies a minimum set of sentences which must be true at those worlds. For example, from the SCS of Figure 2.6 for ω it can be determined that any consistent model must consist of worlds $w_{1'}$, $w_{2'}$, $w_{3'}$, $w_{4'}$, corresponding to w_1, w_2, w_3, w_4 such that $w_{1'} \in W$, $w_{4'} \in W$, $w_{2'} \in W$ or $\neg A$ is true at all worlds accessible from $w_{1'}$, and $w_{3'} \in W$ or $\neg B$ is true at all worlds accessible from $w_{1'}$. $w_{4'} \in P(A \wedge B)$, $w_{4'} \in P(\neg C)$, and $w_{1'} \in P(\neg \omega)$ holds in any consistent relevant model for $\neg \omega$. If $w_{2'} \in W$, then $w_{2'} \in P(A)$, $w_{2'} \in P(C)$; if $w_{3'} \in W$, then $w_{3'} \in P(B)$, $w_{3'} \in P(C)$. If $w_{2'} \notin W$, then $w_{i'} \in P(\neg A)$ for all $w_{i'}$ such that $Ew_{1'}w_{i'}$; if $w_{3'} \notin W$, then $w_{i'} \in P(\neg B)$ for all $w_{i'}$ such that $Ew_{1'}w_{i'}$.

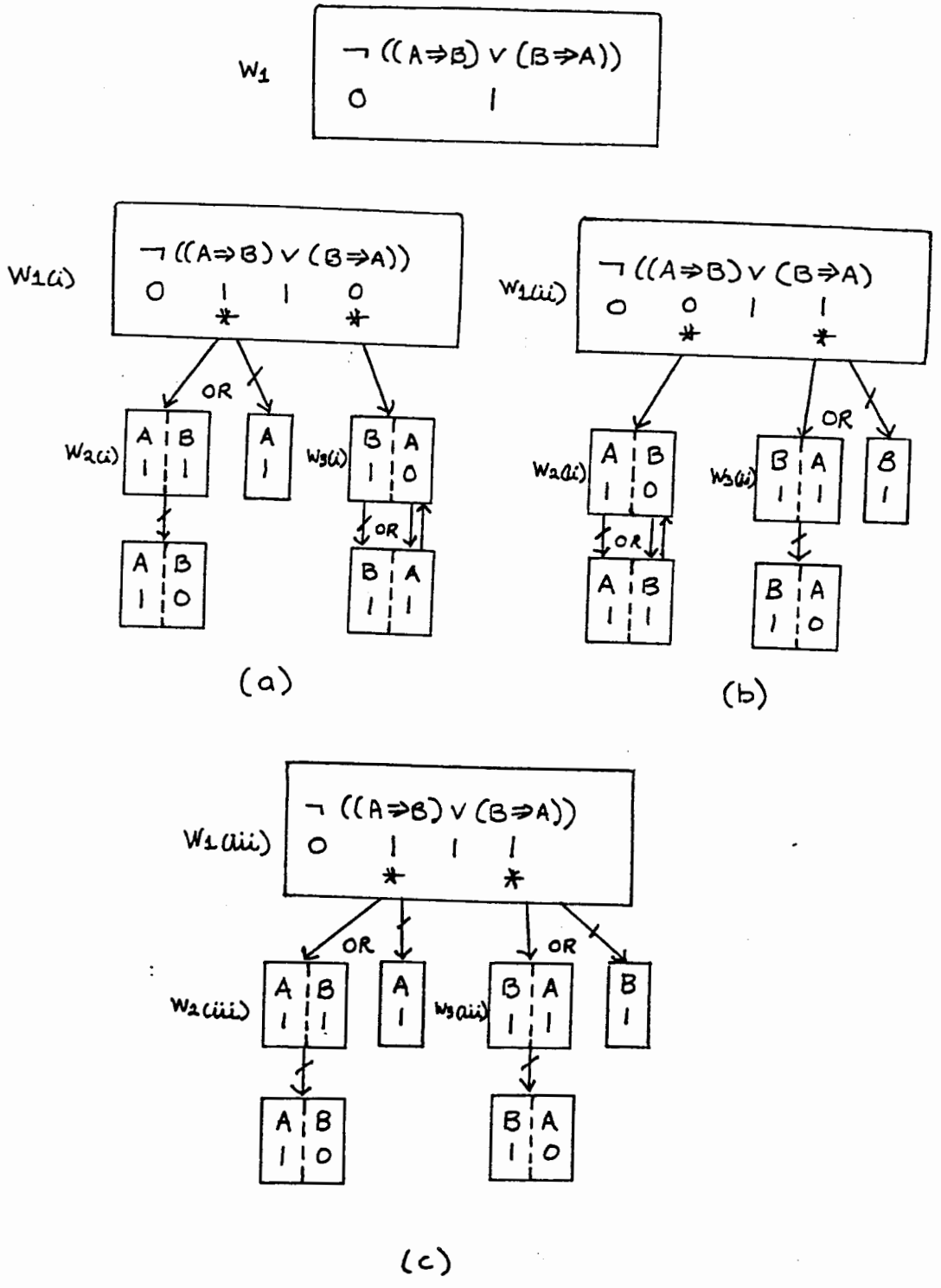


Figure 2.7 Semi-complete-structure for $\omega = ((A \Rightarrow B)\ (B \Rightarrow A))$.

3.3. Generating RTFC Configurations

The goal, then, is to generate all possible relevant RTFC configurations from the SCS. Recall that $\gamma_1 \Rightarrow \gamma_2$ is true at rectangle w_i if *either*:

(1) there exists arrow Aw_iw_j such that γ_1, γ_2 are both assigned 1 at w_j and there exists no rectangle w_k where γ_1 is assigned 1 and γ_2 0 such that Aw_jw_k exists, *or*

(2) for every Aw_iw_j, γ_1 is assigned 0 at w_j.

Therefore consistent model structures may exist in which $w_{j'} \in W$ and others may exist in which $w_{j'} \notin W$. Both of these paths, or branches, along the SCS must be investigated, so we generate new diagrams each one consisting of an alternative path.

A *structural template*, or *template*, is defined as an SCS in which for each $\gamma_1 \Rightarrow \gamma_2$ assigned 1 at w_i, one of the following conditions holds:

(a) Aw_iw_j (and its associated diagram fragment) of truth condition (1) have been removed.

(b) The not-arrow (and its associated unlabelled rectangle) imposing alternative truth condition (2) have been removed.

From the SCS of Figure 2.6, four templates are generated as is evidenced in Figure 2.8. Both Aw_1w_2 and Aw_1w_3 have been removed from the SCS in Figure 2.8(a), Aw_1w_2 and the not-arrow alternative to w_3 have been removed in (b), Aw_1w_3 and the not-arrow alternative to w_2 have been removed in (c), and both not-arrows have been removed in (d). A corresponding set of worlds for W can be determined from each template. For example, the template of Figure 2.8(c) constrains the set of models to those in which $W=\{w_{1'},w_{2'},w_{4'}\}$, where $w_{1'}$, $w_{2'}$, $w_{4'}$ correspond to w_1, w_2, w_4 respectively.

The accessibility relation is, however, RTFC, and the templates comprised of more than two rectangles are not forward-connected. The next step in configuration generation, then, requires the addition of sets of arrows to the templates. The sets of arrows must connect the labelled rectangles of the template in such a way that forward-connectedness and transitivity hold. These properties are expressed in terms of RTFC configuration construction by the following rules:

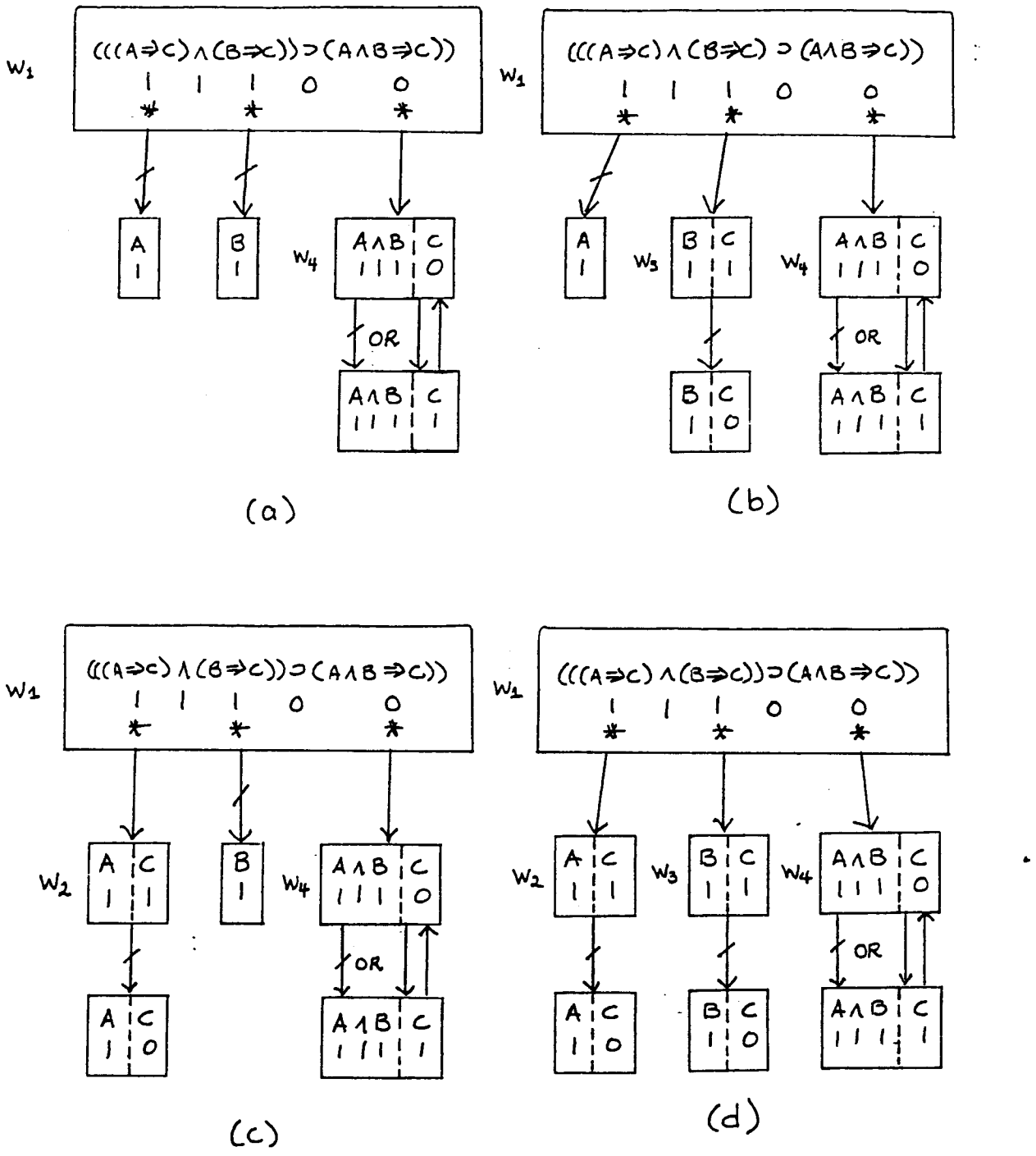


Figure 2.8 Structural templates for the semi-complete structure of Figure 3.6.

FC: If Aw_iw_j and Aw_iw_k both exist, then either (1) Aw_jw_k, (2) Aw_kw_j, or (3) both Aw_jw_k and Aw_kw_j exist.

T: If Aw_iw_j and Aw_jw_k exist, then Aw_iw_k exists.

We can now more formally define a configuration as a template to which a set of arrows has been added so that each pair of rectangles in the template is connected. If the added set of arrows is such that the properties of forward-connectedness and transitivity hold, then the configuration is said to be an RTFC configuration. (Reflexivity is implicit in the template.)

A simple, but inefficient approach to arrow-set generation is to generate all possible arrow sets which connect the template, then test each for transitivity. Note that each pair of rectangles w_1, w_j is already connected by Aw_1w_j. So arrow-set generation is concerned with connecting all labelled rectangles in the template except w_1.[7] There are three different ways of connecting each pair of labelled rectangles by rule FC, and there are $\binom{n}{2}$ pairs of labelled rectangles. Therefore there are $3^{\binom{n}{2}}$ or $O(3^{n^2})$ arrow sets to consider. Each set must be tested for transitivity. If the test fails, the arrow set is rejected. In the worst case the arrow set fully connects the template. Then the arrow set is transitive, so the test for the presence of Aw_iw_k is done for each of the $O(n^3)$ pairs Aw_iw_j and Aw_jw_k. Thus the worst case complexity of the transitivity test for an arrow set is $O(n^4)$, and the complexity of the approach is $O(3^{n^2}n^4)$.

This approach can be avoided, fortunately, if we view a set of worlds in terms of equivalence classes with respect to the accessibility relation E. The RTFC relation imposes a well-ordering on equivalence (EQ) classes of worlds. A set of worlds within an EQ class is, of course, related in such a way that reflexivity, transitivity, and symmetry (and thus forward-connectedness) hold.

The goal is to generate only those arrow sets which relate the rectangles in an RTFC pattern. In model terms, a well-ordering on EQ classes of worlds may consist of from 1 to n EQ classes, where n is the number of worlds in the set W. For example, if $n = 3$ the possibilities are as shown in Figure 2.9. Each diagram of Figure 2.9 is referred to as an equivalence-class template, or EQ template. The outer rectangles of the figure represent EQ classes of worlds with respect to accessibility. The inner rectangles represent

[7] In the case of nested $\Rightarrow$-subformula, a $\Rightarrow$-subformula containing a $\Rightarrow$-subformula, transitive arrows from w_1 to the nested rectangles must be added. These arrows are added to the template.

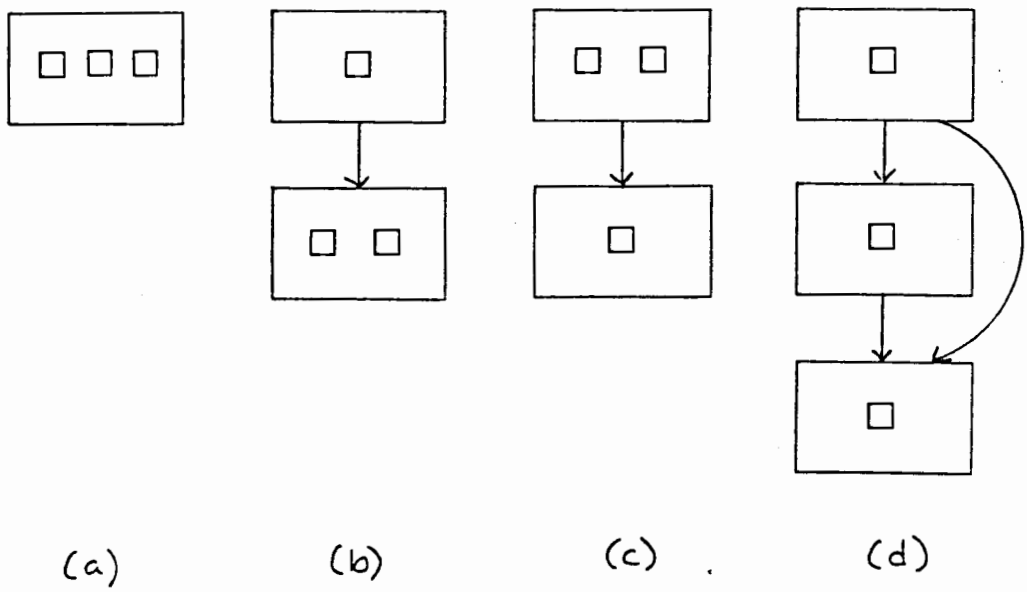


Figure 2.9 System of equivalence-class templates for three worlds.

individual worlds. In Figure 2.9(a) all n worlds are in one EQ class. Figure 2.9(b) and (c) each represent a well-ordering on two EQ classes. The "topmost" EQ class of Figure 2.9(b) is a set consisting of one world, whereas the topmost EQ class of (c) contains two worlds. Figure 2.9(d) represents a well-ordering on n EQ classes, each class consisting of one world.

An arrow from equivalence class EQ_m to equivalence class EQ_p means that for each world w_i in EQ_m, Ew_iw_j for each world in EQ_p. For example, the arrow in the template of Figure 2.10 denotes Ew_2w_4, Ew_2w_5, Ew_3w_4, Ew_3w_5. Ew_2w_3 and Ew_3w_2 hold because w_2 and w_3 are in the same EQ class, and the same applies to Ew_4w_5 and Ew_5w_4.

An arrow from EQ_m to EQ_p means that accessibility "runs downward," i.e., Ew_iw_j but not Ew_jw_i, for w_i in EQ_m and w_j in EQ_p. If Ew_jw_i does hold, then Ew_jw_k must hold for any other w_k in EQ_m by transitivity. Ew_kw_j already holds since w_k is a member of EQ_m. Therefore EQ_m and EQ_p collapse into one EQ class.

So there are EQ-class templates consisting of from 1 to n EQ classes, and the number of worlds within each EQ class varies (Figure 2.9(b) and (c)). Additionally, there are various ways of "fitting" the worlds of W into an EQ template. For example, there are three ways of fitting worlds w_2, w_3, w_4 into the EQ template of Figure 2.9(b) as shown in Figure 2.11.

An EQ template into which a set of rectangle labels (denoting worlds) has been fit is referred to as a *configuration template*, or CF template. Arrow sets are generated from CF templates according to the following algorithm:

Algorithm Arrowset

1. *arrowset* ← {}
2. Repeat
   - 2.1 For each w_i in topmost EQ class do
     - (a) For each w_j, $i \neq j$, in topmost and lower EQ classes do
       - *arrowset* ← *arrowset* ∪ $\{Aw_iw_j\}$
   - 2.2 Remove topmost EQ class

   Until no EQ classes left

To illustrate the algorithm consider the CF template of Figure 2.12. Step 1 initializes *arrowset* to the empty set. The first iteration of step 2.1 yields *arrowset* equal to $\{Aw_3w_4, Aw_3w_5, Aw_3w_2\}$ and the second

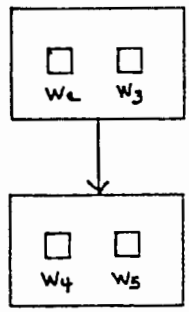


Figure 2.10 An equivalence-class template for four worlds.

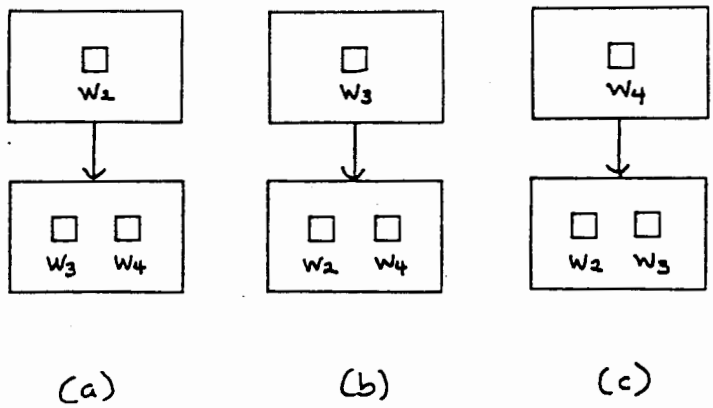


Figure 2.11 Fitting worlds into an equivalence-class template.

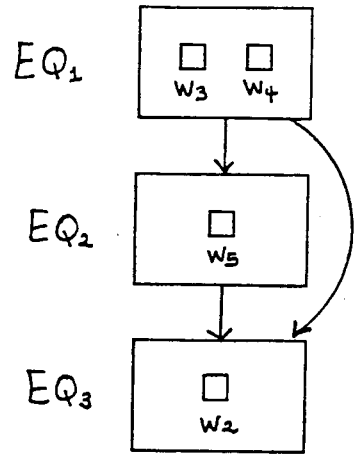


Figure 2.12 A configuration template for four worlds w_2, w_3, w_4, w_5.

arrowset equal to $\{Aw_3w_4, Aw_3w_5, Aw_3w_2, Aw_4w_3, Aw_4w_5, Aw_4w_2\}$. Step 2.2 removes EQ_1, so EQ_2 becomes the topmost EQ class. The for loop of step 2.1 yields *arrowset* equal to $\{Aw_3w_4, Aw_3w_5, Aw_3w_2, Aw_4w_3, Aw_4w_5, Aw_4w_2, Aw_5w_2\}$. EQ_2 is removed, but EQ_3 is the "bottom" EQ class and contains no pairs w_i, w_j. So the set of arrows generated from the CF template of Figure 2.12 is $\{Aw_3w_4, Aw_3w_5, Aw_3w_2, Aw_4w_3, Aw_4w_5, Aw_4w_2, Aw_5w_2\}$.

To return to the example $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset ((A \wedge B) \Rightarrow C))$, we get the templates of Figure 2.9 for three worlds. In order to generate all possible sets of RTFC arrows, we must fit w_2, w_3, w_4 into each EQ template in all combinations. There is only one way of fitting w_2, w_3, w_4 into the template of Figure 2.9(a), i.e., Figure 2.13(a), three possibilities for Figure 2.9(b) as in Figure 2.13(b), three possibilities for Figure 2.9(c) as in Figure 2.13(c), and six possibilities for Figure 2.9(d) as in Figure 2.13(d). The arrow sets generated from these CF templates are shown in Table 2.3.

Table 2.3. Arrow sets generated from the CF templates of Figure 2.13.

A_1	:	$\{Aw_2w_3, Aw_2w_4, Aw_3w_2, Aw_3w_4, Aw_4w_2, Aw_4w_3\}$
A_2	:	$\{Aw_2w_3, Aw_2w_4, Aw_3w_4, Aw_4w_3\}$
A_3	:	$\{Aw_3w_2, Aw_3w_4, Aw_2w_4, Aw_4w_2\}$
A_4	:	$\{Aw_4w_2, Aw_4w_3, Aw_2w_3, Aw_3w_2\}$
A_5	:	$\{Aw_2w_3, Aw_2w_4, Aw_3w_2, Aw_3w_4\}$
A_6	:	$\{Aw_2w_4, Aw_2w_3, Aw_4w_2, Aw_4w_3\}$
A_7	:	$\{Aw_3w_4, Aw_3w_2, Aw_4w_3, Aw_4w_2\}$
A_8	:	$\{Aw_2w_3, Aw_2w_4, Aw_3w_4\}$
A_9	:	$\{Aw_2w_4, Aw_2w_3, Aw_4w_3\}$
A_{10}	:	$\{Aw_3w_2, Aw_3w_4, Aw_2w_4\}$
A_{11}	:	$\{Aw_3w_4, Aw_3w_2, Aw_4w_2\}$
A_{12}	:	$\{Aw_4w_2, Aw_4w_3, Aw_2w_3\}$
A_{13}	:	$\{Aw_4w_3, Aw_4w_2, Aw_3w_2\}$

The work accomplished up to this point for $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$ is summarized in the structure hierarchy of Figure 2.14. The method begins by assigning 0 to $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$ (level 1 of the hierarchy). The semi-complete structure for ω assigned 0 (Figure 2.6) is

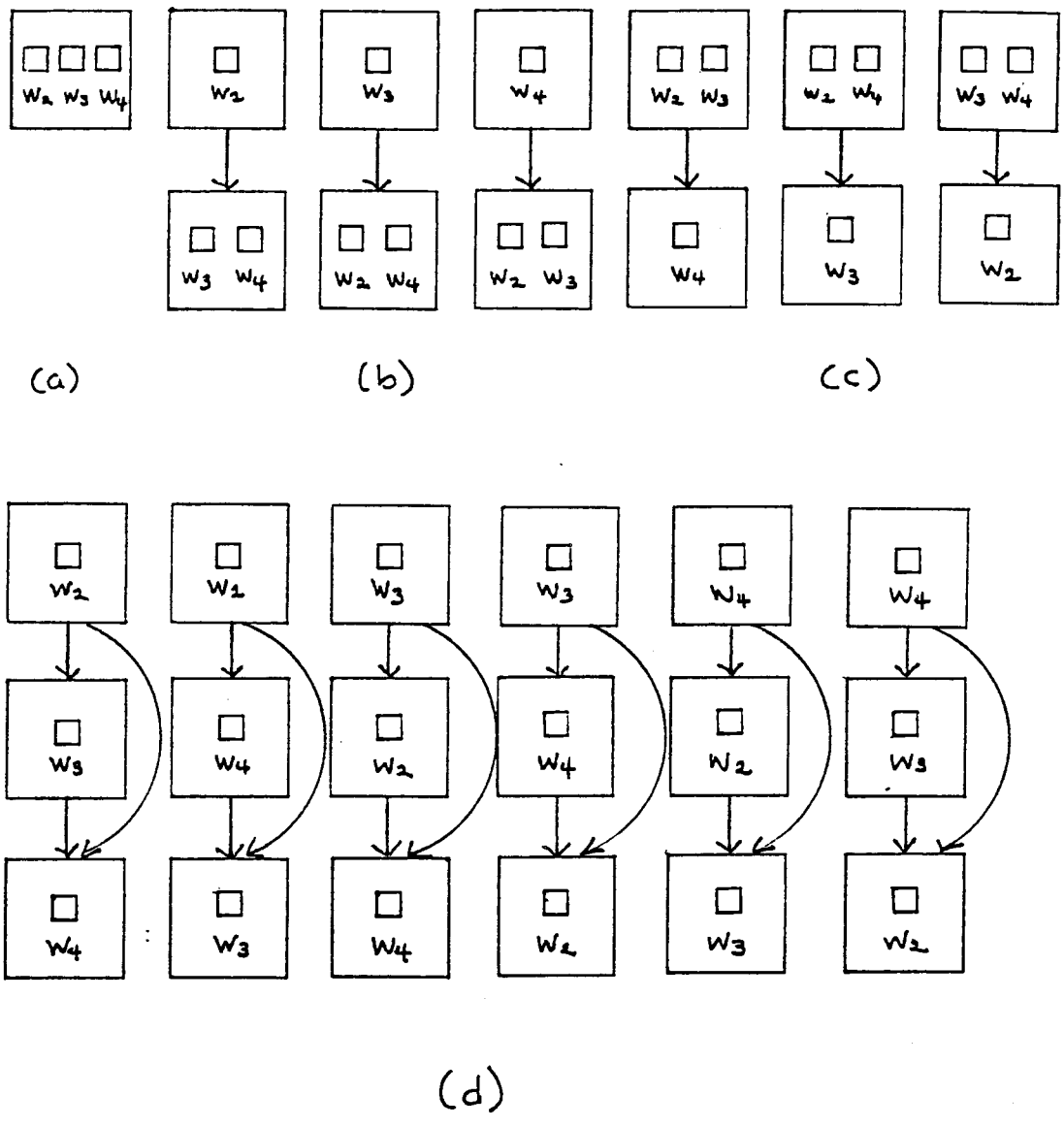


Figure 2.13 The configuration templates generated from the equivalence-class templates of Figure 3.9.

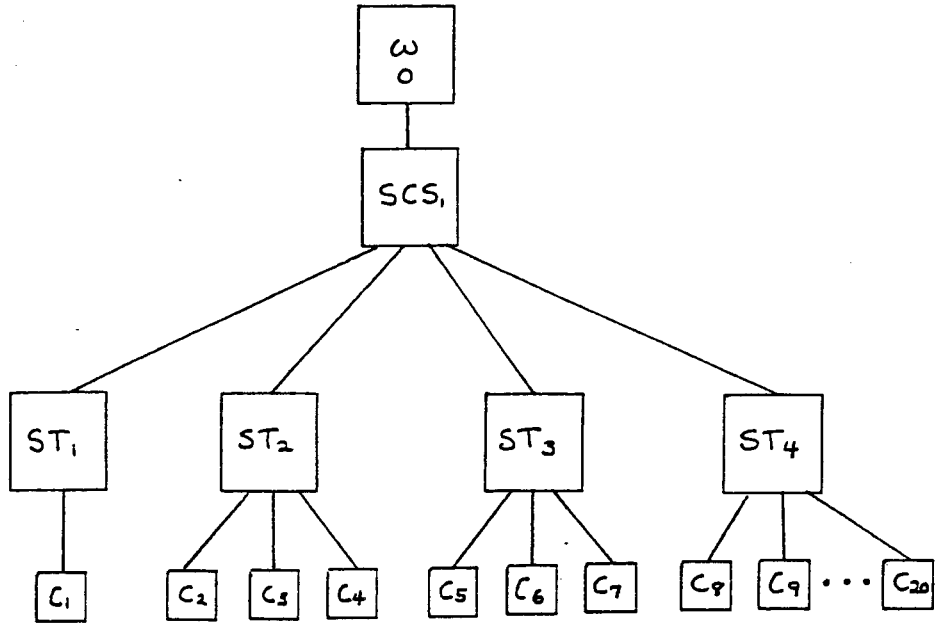


Figure 2.14 A hierarchy of structures used in the construction of relevant RTFC configurations for $\omega = (((A \Rightarrow B) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$.

constructed (level 2 of the hierarchy). More than one SCS is constructed for wffs containing β-subformula, so level 2 may contain more than one node. From the SCS we obtain alternative structural templates (of Figure 2.8) ST_1, ST_2, ST_3, ST_4 (level 3 of the hierarchy). By applying the arrow set generator described previously we get the relevant RTFC configurations $C_1, ..., C_{20}$ (level 4). $C_8, ..., C_{20}$ contain arrow sets $A_1, ..., A_{13}$ respectively, of Table 2.3.

These RTFC configurations represent all relevant models in which ω is false. If any one is consistent, ω is invalid; otherwise ω is valid. Therefore the next step involves testing configurations for consistency.

3.4. Testing Configuration Consistency

We can now formally define inconsistency within a configuration. Any rectangle in which both 0 and 1 have been assigned to the same subformula is an *inconsistent rectangle*. An arrow Aw_iw_j is consistent if the conditions on accessibility from w_i are consistent with the conditions which hold at w_j. Recall that the conditions on accessibility from w_i are the negations of the subformula in unlabelled rectangles connected to w_i by not-arrows. This means that Aw_iw_j is consistent iff the conditions on accessibility from w_i and the conditions at w_j, i.e., the sentences true at w_j, are satisfiable.

Thus we can obtain a formula Aw_iw_j' denoting an arrow Aw_iw_j in which Aw_iw_j' is the conjunction of the set of sentences true at w_j and the negations of the subformula in unlabelled rectangles connected to w_i by not-arrows. Aw_iw_j' is inconsistent if both 0 and 1 have been assigned to the same subformula of Aw_iw_j'. An *inconsistent arrow* is an arrow Aw_iw_j for which Aw_iw_j' is inconsistent. For example, the structural template of Figure 2.8(b) contains an inconsistent arrow Aw_1w_4. The conditions on accessibility from w_1 require that $\neg A$ be assigned 1 at all accessible rectangles, but A is assigned 1 at w_4.

A generated arrow set may contain more than one arrow entering the same rectangle, say Aw_iw_j, Aw_kw_j. It may also be the case that Aw_iw_j is consistent and Aw_kw_j is consistent, but together the two are *mutually inconsistent*. This situation arises in the configuration of Figure 2.15. Here $Aw_2w_4' = \neg(A \wedge \neg B) \wedge A \wedge \neg C$, $Aw_3w_4' = \neg(B \wedge \neg C) \wedge A \wedge \neg C$. Aw_2w_4' is satisfiable by the assignment of 1 to A and B and 0 to C, and this is the *only* assignment which satisfies Aw_2w_4'. Aw_3w_4' is satisfied

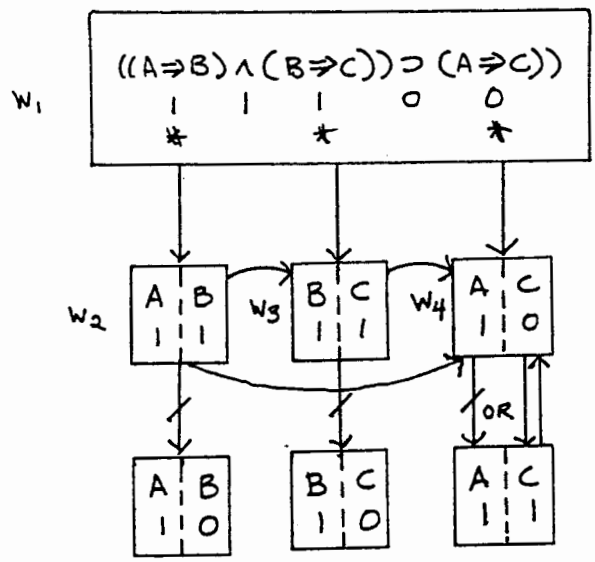


Figure 2.15 A configuration containing mutually inconsistent arrows.

by the assignment of 1 to A and 0 to B and C, and this is the *only* assignment which satisfies Aw_3w_4'. So Aw_2w_4' is consistent only if B is assigned 1 at rectangle w_4, and Aw_3w_4' is consistent only if B is assigned 0 at rectangle w_4. The two obviously cannot consistently coexist.

Note that in the case of a rectangle w_i created by the rule for new worlds from a false $\Rightarrow$ (e.g., w_4 in Figure 2.15), there are alternative conditions on accessibility from w_i. Aw_iw_j is consistent if the negation of the forbidden value assignments are consistent with the conditions at w_j, *or* Aw_jw_i coexists. If Aw_jw_i coexists, then the truth conditions are satisfied regardless of the value assignments at w_j. Thus $Awwj'$ is defined as the conjunction of each formula Aw_iw_j' such that Aw_iw_j is an arrow of the configuration except where Aw_jw_i coexists and w_i was created due to a false $\Rightarrow$. $Awwj'$ can be simplified by factoring out the conditions at w_j. Then $Awwj'$ becomes the conjunction of the conditions which hold at w_j and the conditions on accessibility from w_i, for each w_i such that Aw_iw_j is an arrow of the configuration (except where Aw_jw_i coexists and w_i was created due to a false $\Rightarrow$). $Awwj'$ is inconsistent if both 0 and 1 have been assigned to the same subformula of $Awwj'$. A *mutually inconsistent set of arrows* is a set of arrows $\{Aw_iw_j, ..., Aw_kw_j\}$ for which $Awwj'$ is inconsistent.

An *inconsistent configuration* is defined as follows:

(1) A configuration containing an inconsistent rectangle is inconsistent.

(2) A configuration containing an inconsistent arrow is inconsistent.

(3) A configuration containing a mutually inconsistent set of arrows is inconsistent.

(4) A configuration containing a set of arrows which connects rectangles in such a way that forward-connectedness or transitivity fails is inconsistent.

If we employ the arrow-set generation approach which makes use of the well-ordering on EQ classes of labels we do not have to be concerned with type (4) inconsistency. Every configuration generated is RTFC.

Consider once again the structural template of Figure 2.8(b). Every configuration generated from this template contains the template plus a set of arrows embedded into the template to connect each pair of rectangles. Therefore every configuration generated from the template of Figure 2.8(b) is inconsistent

because each contains the inconsistent arrow Aw_1w_4. Likewise if the structural template contained an inconsistent rectangle, every configuration generated from the template is inconsistent. Thus an *inconsistent structural template* is defined as follows:

(1) A structural template containing an inconsistent rectangle is inconsistent.

(2) A structural template containing an inconsistent arrow is inconsistent.

(3) A structural template from which every generated configuration is inconsistent, is inconsistent.

To improve efficiency, arrow and rectangle consistency tests are performed on each structural template before arrow-set generation is begun. If a test fails, the structural template is inconsistent, and no configurations are generated from this template. The branch of the structural hierarchy containing this template is in effect closed (in the sense of a closed tableau branch). Using this technique on our example $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$, we get the structural hierarchy of Figure 2.16, a modification of Figure 2.14. A structural template ST_i is also inconsistent if every configuration generated from ST_i is inconsistent.

A semi-complete structure each of whose structural templates is inconsistent, is inconsistent. If for any wff ω rectangle w_1 is an explicitly inconsistent rectangle, then ω is valid since rectangle w_1 contains exactly one sentence of N, ω, and it is assigned 0. In this case ω is proven before the SCS is built. ω is valid iff one of the following conditions hold:

(1) Rectangle w_1 is explicitly inconsistent.

(2) Every SCS for ω is inconsistent.

All that remains is to test each configuration for consistency. A configuration consistency test is accomplished by the following algorithm:

1. Sort arrows of the configuration according to the rectangle entered
2. Repeat
   - 2.1. Determine $Awwj'$ for the set of arrows entering w_j
   - 2.2. Test $Awwj'$

   Until some $Awwj'$ is inconsistent or all $Awwj'$ tested
3. If an inconsistency is found return inconsistent
   Else return consistent

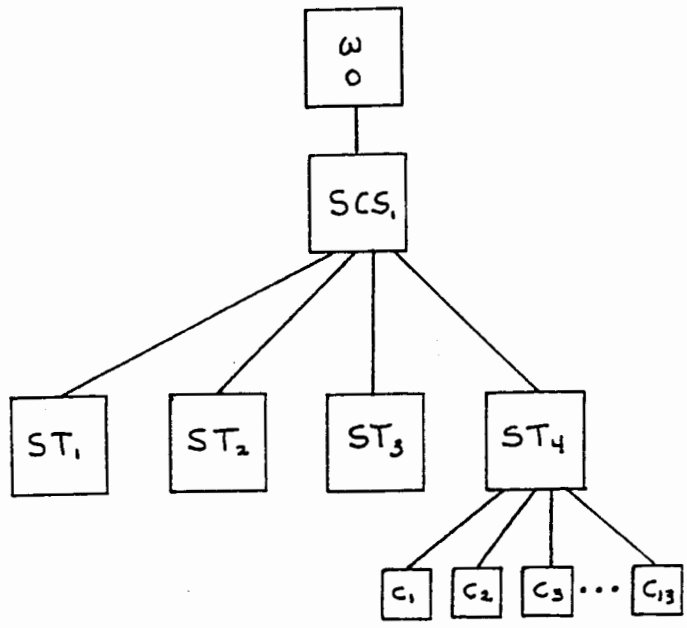


Figure 2.16 The modified structural hierarchy for $\omega = (((A \Rightarrow B) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$.

We do not need to test each individual arrow additionally. If Aw_iw_j' is inconsistent then Aww_j' is inconsistent because Aw_iw_j' is a subformula of Aww_j'. If any configuration consistency test returns consistent, then algorithm NTP1 returns invalid. If all configuration tests return inconsistent, then NTP1 returns valid.

The configuration consistency test is repeated for every configuration in the case that ω is valid. The Aww_j' are the same in many configurations. In order to improve the efficiency of configuration testing, an alternative approach is developed. Each arrow is tested only once and the results kept in a table of forced values. We want to know what must be true at w_j in order for Aw_iw_j to be consistent. Aw_iw_j' is tested for consistency, i.e., Aw_iw_j' is assigned 1 and the consequential values determined. Assignments to propositional variables, $\Rightarrow$-subformula, and β-subformula are stored in the table. The table of forced values for our example $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$ is given in Table 2.4.

Table 2.4. Forced values for $\omega = ((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset ((A \wedge B) \Rightarrow C)$.

Arrows	A	B	C
Aw_3w_2	1		1
Aw_4w_2	1	0	1
Aw_2w_3		1	1
Aw_4w_3	0	1	1
Aw_2w_4	1	1	0
Aw_3w_4	1	1	0

Arrow Aw_4w_2 forces the assignment of 0 to B at w_2, but Aw_3w_2 is consistent if B is assigned 0 or 1 at w_2. Arrows Aw_2w_4 and Aw_3w_4 are each inconsistent arrows, so no consistent configuration can contain either of these arrows. Information about constraints imposed upon any consistent configuration is stored in a table of arrow constraints (see Table 2.5).

Table 2.5. Arrow constraints for $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$.
Aw_2w_4 cannot occur
Aw_3w_4 cannot occur

The first step in configuration testing, then, requires a check to see if any of the arrow constraints are contradicted. Arrow sets A_1, A_2, A_3 of Table 2.5 each contain Aw_2w_4 and Aw_3w_4 contradicting both constraints. (Of course, a configuration is inconsistent even if only one constraint is contradicted.) However A_4 generates the configuration of Figure 2.17. Aw_4w_2 is consistent since B is assigned 0 at w_2, and Aw_4w_3 is consistent since A is assigned 0 at w_3. Each arrow is consistent, all arrows are mutually consistent, and all rectangles are consistent. Thus $\neg\omega$ is satisfiable, and ω is invalid. Table 2.6 serves as a counterexample.

In this N-model $W = \{w_1, w_2, w_3, w_4\}$, $E = \{Ew_1w_1, Ew_2w_2, Ew_3w_3, Ew_4w_4, Ew_1w_2, Ew_1w_3, Ew_1w_4, Ew_4w_2, Ew_4w_3, Ew_2w_3, Ew_3w_2\}$. The function P is defined by the table itself. For example, $P(A) = \{w_4, w_2\}$. Values for the propositional variables A, B, C have been left blank at w_1. They can consistently be assigned 0 or 1 at w_1 in any combination. To have a definite rule, assign 1 to such variables.

Note that as soon as a consistent configuration is found, the invalidity of ω is determined. We do not need to look at any other configurations. Therefore an obvious improvement to algorithm NTP1 is to test configurations as they are generated rather than generating all configurations and then testing. Steps 3 and 4 of NTP1 then become:

3. Repeat
   Generate an RTFC configuration
   Test the configuration
   Until all RTFC configurations tested or a consistent configuration found

If we "bend" the rule for alternatives, we can attain a further improvement in efficiency. Apply the rule as usual at w_1. At other rectangles, however, delay applying the rule unless the β-subformula contains a $\Rightarrow$-subformula. In this way we avoid building a system of three SCS which vary only in the assignment of values to β_1 and β_2 at some rectangle w_i, $i \neq 1$. It may be the case that arrows entering w_i force an assignment to β_1 or β_2. The example in the next section contains such a β-subformula.

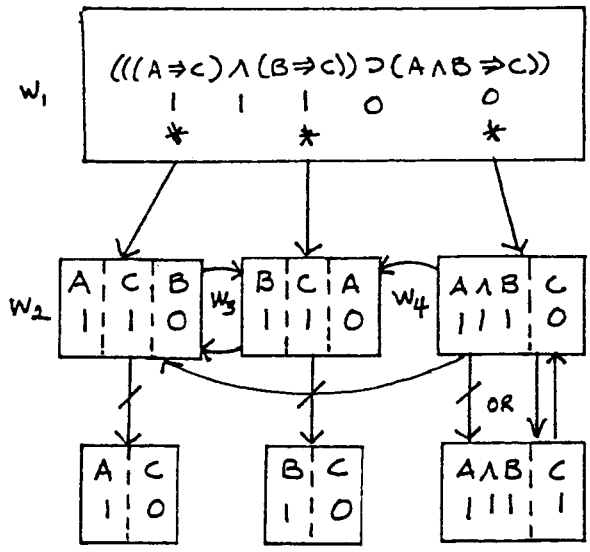


Figure 2.17 A consistent configuration for $\omega = (((A \Rightarrow B) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C))$.

Table 2.6. A counterexample for $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C)$.

worlds	A	B	C	A∨B	A⇒C	B⇒C	A∧B⇒C	(((A⇒C)∧(B⇒C))⊃(A∧B⇒C)
w_1					1	1	0	0
w_4	1	1	0	1			0	
w_2	1	0	1		1			
w_3	0	1	1			1		

Table 2.6. An N-model counterexample for $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \wedge B \Rightarrow C)$. Only relevant value assignments at a world appear in the table. The accessibility relations are depicted in the form of a well-ordering on equivalence classes.

3.5. Algorithm NTP

The improved algorithm, NTP, is presented and illustrated with an example proof.

Algorithm NTP
Input: wff
Output: valid, invalid
1. Initialize the system of N-diagrams
    1.1. Write the wff in a rectangle
    1.2. Label the rectangle w_1
    1.3. Assign 0 to the main operator of the wff
2. Build the semi-complete structure(s)
    2.1. Repeat
        Apply α-rules
        Apply rules for crosses, asterisks, and modified rule for alternatives
        Apply rule for new worlds
    Until rules applied as often as possible
3. Repeat {for each semi-complete structure}
    Generate structural templates from SCS
    Build tables of forced values and arrow constraints
    Repeat {for each structural template}
        Repeat {for each EQ template of $q = 1$ to n classes}
            Repeat {for each EQ template of q classes}
                Repeat {for each CF template}
                    Generate an RTFC configuration
                    Test the configuration
                Until all CF templates processed or ccf
            Until all EQ templates of q classes processed or ccf
        Until all EQ templates processed or ccf
    Until all structural templates processed or ccf
Until all alternative SCS processed or ccf

The abbreviation *ccf* in the algorithm stands for "consistent configuration found." Let us examine the proof of $\omega = (((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow B \wedge C))$ using algorithm NTP. The SCS for ω is given in Figure 2.18. Note that the rule for alternatives is waived at w_4. Arrow Aw_1w_4 is inconsistent in any structural template containing the not-arrow alternative to w_2 or w_3. Thus only one structural template is consistent at this stage, the template of Figure 2.19.

The next step is to test each arrow and store the results in the table of forced values, Table 2.7. The table of arrow constraints stores the implications of the information in the table of forced values. Let us examine each group of arrows sorted according to the rectangle entered. (Table 2.7 is already sorted in this way.) We are looking for mutual inconsistencies. Arrow Aw_2w_4 forces the assignment of 1 to B and 0 to C at w_4, but Aw_3w_4 forces the assignment of 0 to B and 1 to C at w_4. Thus Aw_2w_4 and Aw_3w_4 cannot coexist in

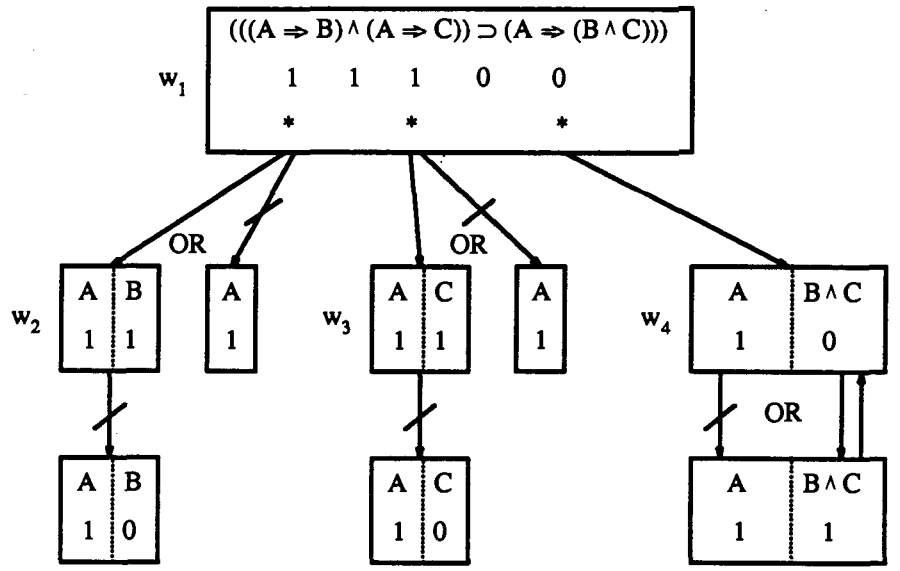


Figure 2.18. Semi-complete structure for $\omega = (((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow B \wedge C))$.

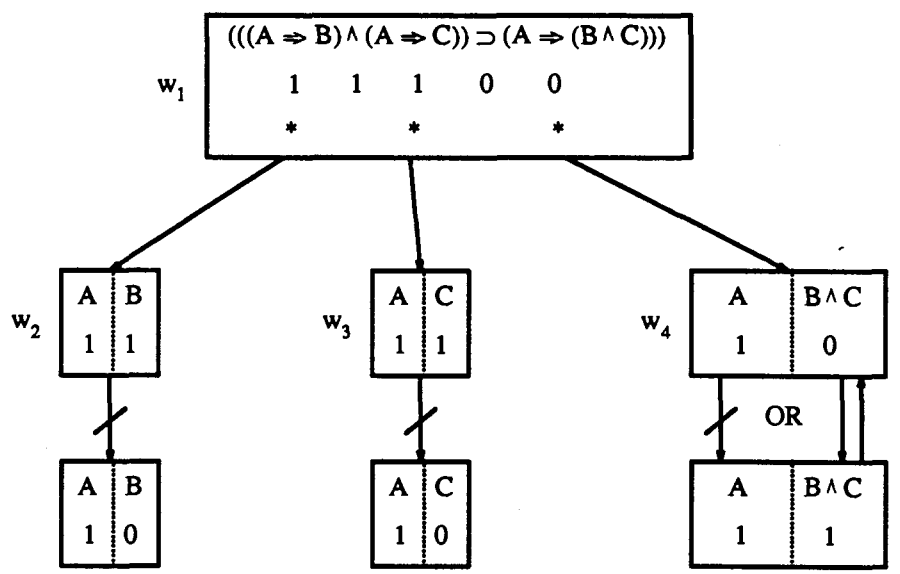


Figure 2.19. Structural template from the semi-complete structure for $\omega = (((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow B \wedge C))$.

any consistent configuration, and this is noted in the table of arrow constraints.

The group of arrows entering w_2 appear at first glance to be incompatible. However, recall that the conditions on accessibility from w_4 (a rectangle created by the rule for a false $\Rightarrow$) are such that if the forbidden conditions do occur at an accessible rectangle w_j, then arrow Aw_jw_4 must also exist. Thus if A, B, C are all assigned 1 at w_2, Aw_4w_2 is still consistent iff Aw_2w_4 co-occurs. So another constraint on any arrow

Table 2.7. Forced values for $\omega = ((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow (B \wedge C))$.

Arrows	A	B	C	B ∧ C
Aw_3w_2	1	1	1	
Aw_4w_2	1	1	0	
Aw_2w_3	1	1	1	
Aw_4w_3	1	0	1	
Aw_2w_4	1	1	0	0
Aw_3w_4	1	0	1	0

set is that if Aw_3w_2 and $Aw4w_2$ occur, then Aw_2w_4 must co-occur. If Aw_3w_2 occurs and Aw_4w_2 does not, then Aw_2w_4 must occur by rule FC. So if Aw_3w_4 occurs then Aw_2w_4 must co-occur, whether or not Aw_4w_2 is present. If Aw_3w_2 and Aw_2w_4 coexist, then Aw_3w_4 must coexist by transitivity.

The same situation arises in the group of arrows entering w_3. We note that if Aw_2w_3 occurs then Aw_3w_4 and Aw_2w_4 must co-occur. These observations are summarized in Table 2.8.

Table 2.8. Arrow constraints for $\omega = (((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow B \wedge C))$.

(1) Aw_2w_4 and Aw_3w_4 cannot coexist

(2) If Aw_3w_2 exists then Aw_2w_4 and Aw_3w_4 must coexist

(3) If Aw_2w_3 exists then Aw_3w_4 and Aw_2w_4 must coexist

Next the arrow sets of Table 2.5 are generated and tested, one by one. Each arrow set contradicts at least one of the arrow constraints. Thus every configuration is inconsistent, and the structural template from which the configurations were generated is therefore inconsistent. Every other structural template has been shown to be inconsistent (before any configurations were generated). Therefore the SCS is inconsistent, and $\neg\omega$ is unsatisfiable. It follows that ω is valid.

Observe that it can be derived from the arrow constraints that no consistent RTFC pattern of arrows is possible:

(a) Aw_3w_2 cannot occur by constraints (1) and (2)

(b) Aw_2w_3 cannot occur by constraints (1) and (3)

(c) One of Aw_2w_3, Aw_3w_2 must occur by forward-connectedness

(d) Contradiction: (c) and (a) and (b)

A contradiction can be derived from the set of constraints on RTFC patterns of arrows. But these constraints were derived from the table of value assignments which are forced in order to make each arrow consistent. No RTFC set of arrows can satisfy these constraints. Thus no consistent RTFC set of arrows is possible. It follows that no consistent RTFC configuration is possible, and the structural template containing this set of rectangles is inconsistent. All other structural templates are inconsistent, so ω is valid. The validity of ω is thus determined without generating any configurations. In the case of an invalid ω, no such derivation is, of course, possible.

3.6. Summary

To summarize the procedure, we begin by writing the wff to be proven, ω, in a rectangle labelled w_1 and placing a 0 under the main operator. In each rectangle apply α-rules first. Rectangles other than w_1 are constructed according to the following rules:

(1) rule for crosses,

(2) modified rule for alternatives,

(3) rule for asterisks,

(4) rule for new worlds.

A system of N-diagrams for ω is semi-complete when the rules have been applied as often as possible. The semi-complete structure may contain ORs and not-arrows alternative to labelled rectangles. So all possible structural templates are generated from the SCS. A preliminary test is performed on each template. A template containing an inconsistent arrow or an inconsistent rectangle is inconsistent. No configurations are generated from an inconsistent structural template.

If at least one consistent template remains, then construction of the table of forced values begins. Rows of the table are labelled with the set of arrows which fully connects the labelled rectangles of the SCS.[8] Column headings are the relevant subformulas. Each arrow is tested and forced values placed in the table. These forced values are the value assignments to subformula which must hold at w_j in order for Aw_iw_j to be consistent.

The next step is to look for mutual inconsistencies among arrows entering the same rectangles. These inconsistencies must be avoided in our search for a consistent RTFC configuration. The restrictions imposed by the mutual inconsistencies are stored in the table of arrow constraints.

An RTFC set of arrows is generated and tested for compatibility with the set of arrow constraints. If incompatible, that set of arrows is rejected and the next set generated and tested until all sets have been tested or a compatible set found. A compatible set of arrows when added to the structural template forms a consistent configuration for $\neg\omega$. ω is therefore invalid. If no compatible set is found, then $\neg\omega$ is unsatisfiable and ω valid.

If a consistent configuration is found, the corresponding N-model is defined by:

(1) $W = \{w_i |\ w_i$ is a rectangle label in the configuration$\}$

(2) $E = \{Ew_iw_j |\ Aw_iw_j$ is an arrow in the configuration$\}$

(3) P = table of world values.

The table of world values is constructed in the following way. Column headings are the relevant subformula at rectangles; row labels are the rectangle labels of the configuration. Fill in value assignments to subformulas according to the following algorithm:

[8] To improve efficiency omit arrows Aw_iw_1 and include Aw_1w_j, including $j=1$, only when the structural template contains not-arrows imposing conditions on accessibility from w_1.

1. For each arrow Aw_iw_j in the table of forced values do

   If w_i was created by rule for a false $\Rightarrow$ and Aw_jw_i coexists

   Then ignore Aw_iw_j

   Else add value assignments from table of forced values to world values

2. If Aw_1w_1 is not in table of forced values then add initial assignment of

   0 to ω and consequential values to relevant subformulas of ω

   from rectangle w_1

The corresponding N-model serves as a counterexample, and ω is invalid. Table 2.6 represents the N-model corresponding to the consistent configuration of Figure 2.17, for example.

Chapter 3. Correctness Proof

The proof of correctness presented in this chapter is inspired by the proofs for some conditional logics in [Burgess 81], for *T*-, *S*4-, and *S*5-diagrams in [Hughes and Cresswell 68], for temporal logics in [Rescher and Urquhart 71], and for tableaux systems in [Smullyan 68]. The model theory of N is sound and complete with respect to the proof theory. The proof is presented in [Delgrande 87]. Therefore if it can be shown that the algorithm is correct with respect to the model theory, then it follows that the algorithm is sound and complete with respect to the proof theory. Briefly, the correctness proof demonstrates that algorithm NTP is consistent (section 1), and sound (section 2) and complete (section 3) with respect to the model theory.

1. Consistency

The goal is to establish that the algorithm is consistent in the sense that no formula and its negation are both provable. The proof is based upon two lemmas and the completeness result of section 3. It is demonstrated that the algorithm terminates in a finite number of steps (lemma 1), and is unambiguous (lemma 2). By the completeness result every wff provable by the algorithm (referred to as an NTP-valid wff) is valid in the model theory and therefore a theorem of the axiomatic system. However, no formula and its negation can both be theorems. Thus it remains to prove lemmas 1 and 2. The completeness result is presented in section 3.

Lemma 1. The method of N-diagrams terminates in a finite number of steps.

Proof. The method consists of two main parts: (1) building a semi-complete system of N-diagrams, and (2) generating and testing all possible sets of arrows which, when added to the semi-complete structure form RTFC configurations. Thus part (1) terminates when the SCS is constructed. An SCS is a system of N-diagrams in which the rules of the method have been applied as often as possible so that:

(a) for every rectangle containing a †, i.e., a β-subformula, the appropriate alternative diagrams have been constructed according to the rule for alternatives,

(b) for every rectangle not containing a †, new worlds have been created for each asterisk under a $\Rightarrow$ operator according to the rule for new worlds.

The procedure terminates earlier in the event that the initial rectangle w_1 is shown to be inconsistent.

The number of rectangles created due to the application of the rule for alternatives is finite, 3^n at most, where n is the number of rectangles containing a †. This assertion follows from the facts that:

(1) A finite formula contains a finite number of symbols.

(2) At most three alternative rectangles are created for each rectangle in which a † occurs.

(3) Each alternative rectangle contains the same value assignments as the rectangle containing the † plus one alternative assignment to the β-subformula.

The number of rectangles created due to the application of the rules for new worlds is also finite. In order to demonstrate this assertion it is convenient to define the *γ-degree of a formula*. The γ-degree of a formula refers to the number of occurrences of the $\Rightarrow$ symbol within the formula. Specifically:

(1) A variable is of degree 0, as is any formula containing no $\Rightarrow$ operator.

(2) If γ_1, γ_2 are of degrees n_1, n_2, respectively, then $\gamma_1 \Rightarrow \gamma_2$ is of degree $n_1 + n_2 + 1$.

The γ-degree of a rectangle is the sum of the γ-degrees of the formulas within the rectangle. Then if rectangle w_j is created due to an asterisk under a $\Rightarrow$ operator at w_i, the γ-degree of w_j is at least one less than the γ-degree of w_i. This follows from the rule for new worlds because the conditions at rectangle w_j are the antecedent and the consequent (or its negation) of a subformula $\gamma_1 \Rightarrow \gamma_2$ at w_i. If n_1, n_2 are the γ-degrees of γ_1, γ_2 then the γ-degree of w_i is at least $n_1 + n_2 + 1$, whereas the γ-degree of w_j is $n_1 + n_2$. Eventually rectangles containing only propositional formulas are reached. These rectangles contain no asterisks, so the rules for new worlds can no longer be invoked.

Part (2) of the method generates and tests configurations. The procedure terminates if a consistent configuration is found or all configurations have been tested. The given wff is valid iff every RTFC configuration is inconsistent. Even in the case of a valid wff requiring generation of all possible configurations, the number of configurations is finite. Suppose the naive generator is used which returns sets of arrows ensuring the configuration is connected but not necessarily transitive. The number of sets of

arrows generated is large but finite, $3^{\binom{n+1}{2}}$, where $n+1$ is the number of rectangles in the SCS. (There are three ways of connecting each of $\binom{n+1}{2}$ pairs of rectangles.) Thus the method of N-diagrams always terminates in a finite number of steps.

Lemma 2. The method of N-diagrams yields an unambiguous result.

Proof. For any wff ω each SCS is either shown to be consistent or shown to be inconsistent. An SCS is inconsistent if one of the following conditions holds:

(1) Rectangle w_1 is explicitly inconsistent.

(2) Each structural template is inconsistent, i.e., every RTFC configuration is inconsistent.

A consistency test fails iff both 0 and 1 are assigned to the same subformula. Thus each consistency test yields an unambiguous result. The method can be applied to any wff ω of N. ω is placed in rectangle w_1 and assigned 0. The rules of the method are applied to build the SCS. In cases (1) or (2) the procedure terminates and returns valid. Otherwise the procedure terminates earlier, i.e., as soon as a consistent RTFC configuration is found.

2. Soundness

The model theory of N has been shown to be sound and complete with respect to the proof theory in [Delgrande 87]. Thus every wff valid in the model theory is a theorem of N. If in addition every theorem of N is NTP-valid, then it follows that every valid wff is NTP-valid. The soundness proof demonstrates this last point.

Theorem 1. Every valid wff is NTP-valid.

It is first proven that every theorem of N is NTP-valid (lemma 3). The proof of theorem 1 follows by transitivity from the soundness and completeness of the model theory with respect to the proof theory and lemma 3.

Lemma 3. Every theorem of N is NTP-valid.

Proof. The proof is straightforward but tedious. It must be shown that each axiom of the system is valid and that each rule preserves validity. An alternative axiomatic basis for N is given by:

A0 $A \Rightarrow A$

A1 $((A \Rightarrow B) \wedge (B \Rightarrow A)) \supset ((A \Rightarrow C) \equiv (B \Rightarrow C))$

A2 $((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \vee B \Rightarrow C)$

R0 If $\vdash (B_1 \wedge \cdots \wedge B_n) \supset B$ then $\vdash (A \Rightarrow B_1 \wedge \cdots \wedge A \Rightarrow B_n) \supset A \Rightarrow B$

2.1. Proof of Axiom A0

The SCS for A0 is shown in Figure 3.1. Rectangle w_2 is explicitly inconsistent, thus the template is inconsistent. There is only one template, so the SCS is inconsistent. It follows that $A \Rightarrow A$ is NTP-valid.

2.2. Proof of Axiom A1

The SCS for A1 is given in Figure 3.2. If both alternative rectangles $w_{1(i)}$ and $w_{1(ii)}$ are inconsistent then w_1 is inconsistent, and A1 is NTP-valid. The structural templates for Figure 3.2(a) are given in Figure 3.3. Templates (a)-(g) are all inconsistent because:

(1) w_5 and Aw_1w_5 must be in every template.

(2) Aw_1w_5 is inconsistent in any template containing the not-arrow alternative to w_3.

(3) Therefore w_3 and Aw_1w_3 must be in every template.

(4) But Aw_1w_3 is inconsistent in any template containing the not-arrows alternative to w_2 or w_4.

Template (h) is consistent at this stage, so the next step is to build the table of forced values (Table 3.1) and the table of arrow constraints (Table 3.2). It can be deduced from the table of constraints that no consistent configuration is possible:

(a) Aw_2w_5 and Aw_4w_5 must exist by (1) and (3) and forward-connectedness.

(b) Aw_3w_4 must exist by (2) and (4) and forward-connectedness.

(c) Aw_3w_5 must exist by (a) and (b) and transitivity.

(d) Aw_3w_5 cannot exist by (a) and (4).

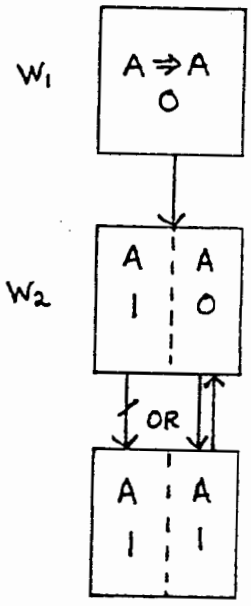


Figure 3.1 Semi-complete structure for A0.

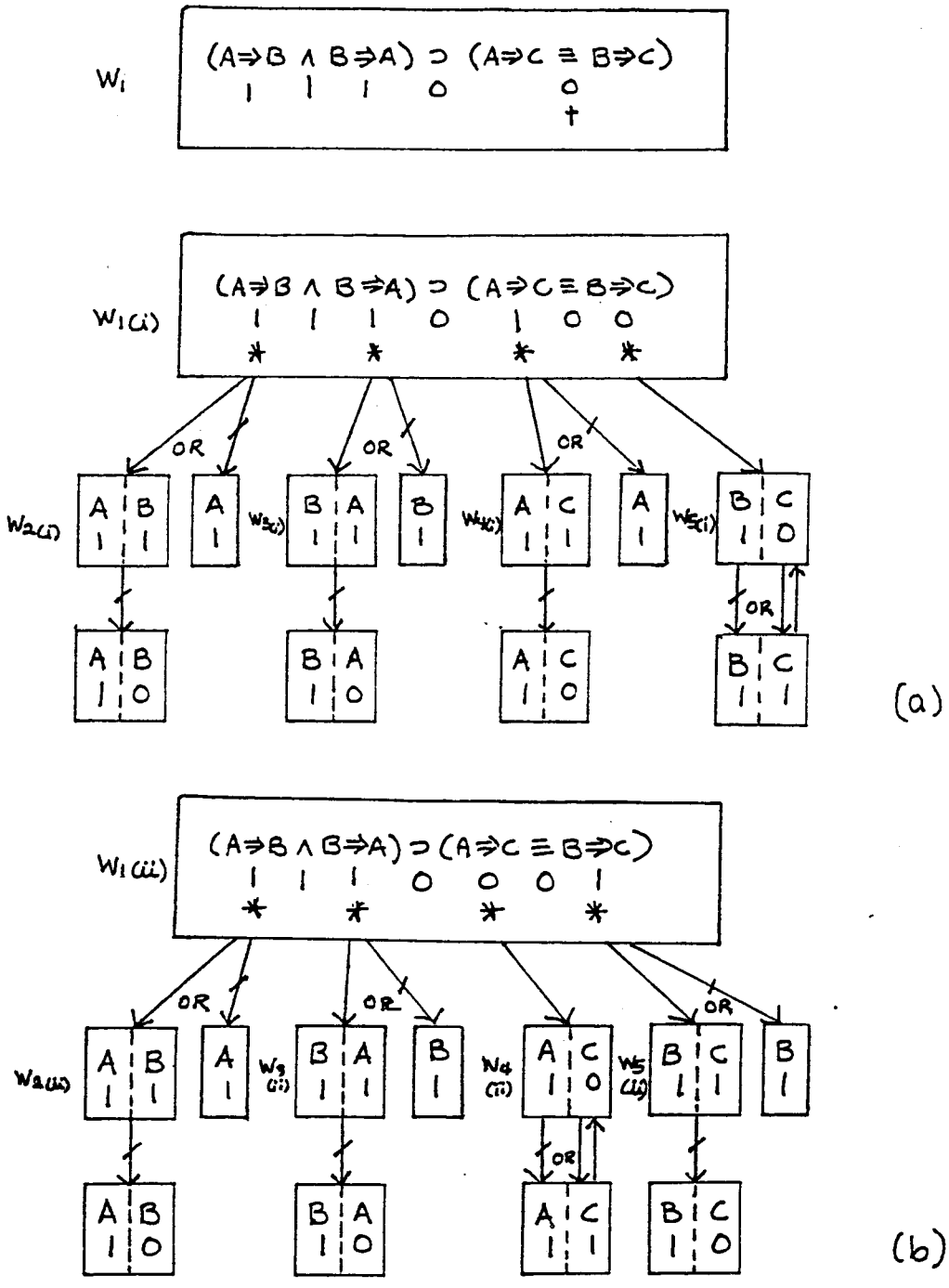
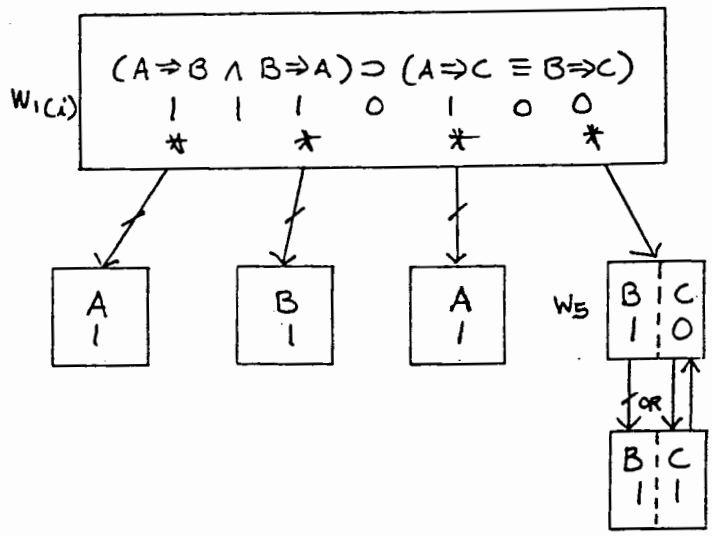
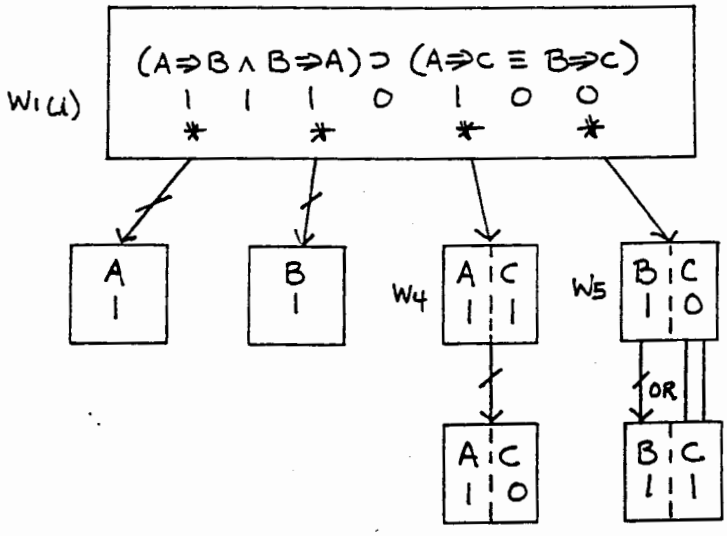


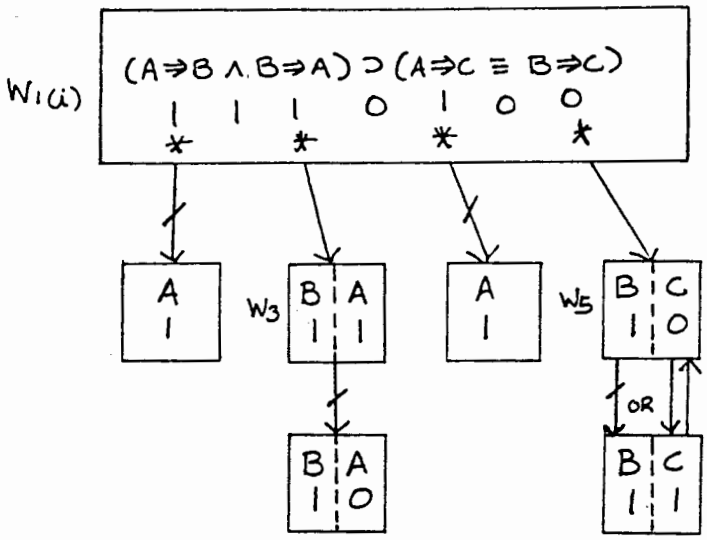
Figure 3.2 Semi-complete structure for A1.


W1(i)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 1 0 0
A 1
B 1
A 1
W5
B C
1 0
OR
B C
1 1

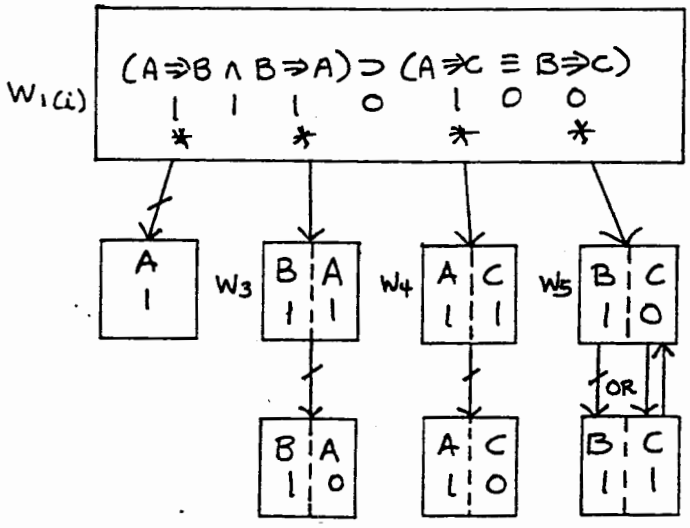
(a)


W1(i)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 1 0 0
A 1
B 1
W4
A C
1 1
W5
B C
1 0
OR
A C
1 0
B C
1 1

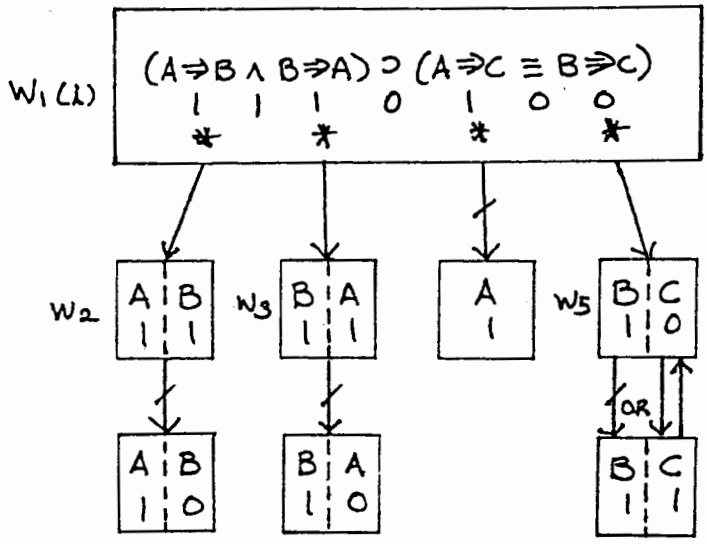
(b)


W1(i)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 1 0 0
A 1
W3
B A
1 1
A 1
W5
B C
1 0
OR
B A
1 0
B C
1 1

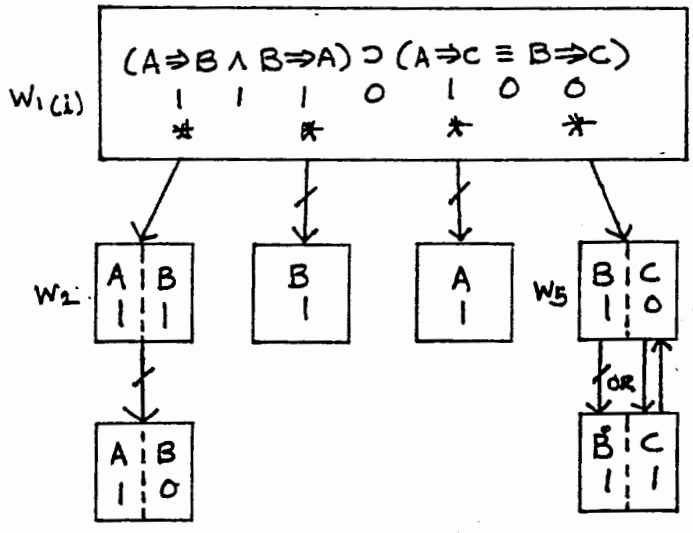
(c)


W1(i)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 1 0 0
A 1
W3
B A
1 1
W4
A C
1 1
W5
B C
1 0
OR
B A
1 0
A C
1 0
B C
1 1

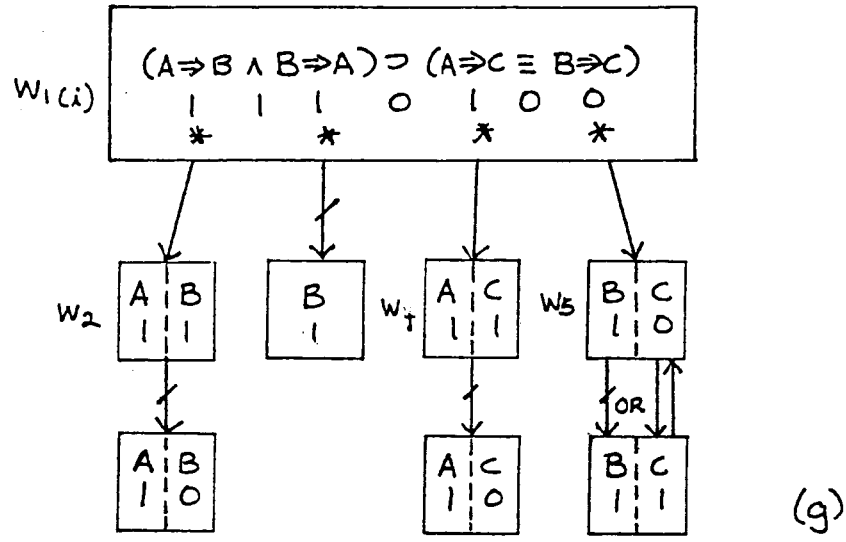
(d)


W1 (1)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 1 0 0
W2
A B
1 1
A B
1 0
W3
B A
1 1
B A
1 0
A
1
W5
B C
1 0
OR
B C
1 1

(e)


W1 (1)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 1 0 0
W2
A B
1 1
A B
1 0
B
1
A
1
W5
B C
1 0
OR
B C
1 1

(f)



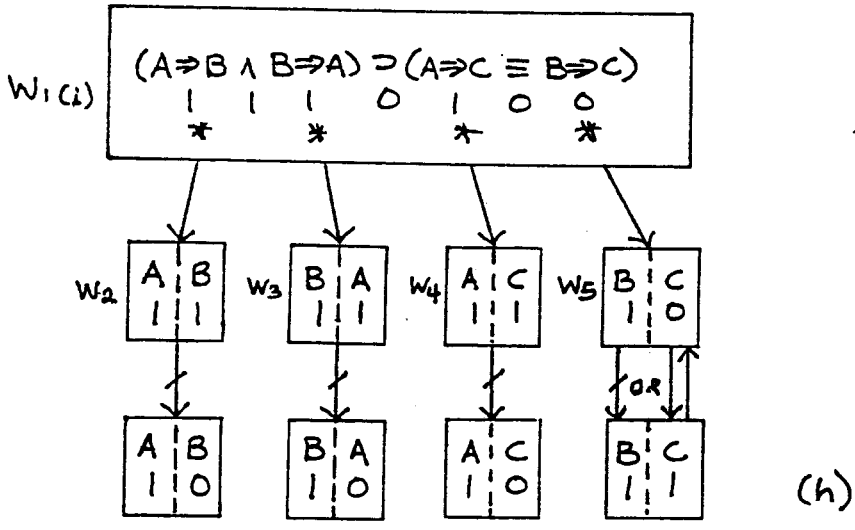


Figure 3.3 Structural templates for Figure 4.2(a).

Table 3.1. Forced values for SCS(a) for A1.

Arrows	A	B	C
Aw_3w_2	1	1	
Aw_4w_2	1	1	1
Aw_5w_2	1	1	0
Aw_2w_3	1	1	
Aw_4w_3	1	1	1
Aw_5w_3	1	1	0
Aw_2w_4	1	1	1
Aw_3w_4	1		1
Aw_5w_4	1	0	1
Aw_2w_5		1	0
Aw_3w_5	1	1	0
Aw_4w_5	0	1	0

Table 3.2. Arrow constraints for SCS(a) of A1.

(1) If Aw_4w_2 exists then Aw_2w_5 and Aw_4w_5 must coexist

(2) If Aw_4w_3 exists then Aw_3w_5 and Aw_4w_5 must coexist

(3) If Aw_2w_4 exists then Aw_4w_5 and Aw_2w_5 must coexist

(4) Aw_3w_5 and Aw_4w_5 cannot coexist

A contradiction is reached in (c) and (d). No configuration can satisfy these constraints, and the SCS is therefore inconsistent.

Next consider the SCS of Figure 3.2(b). The structural templates are given in Figure 3.4. Templates (a)-(g) are all inconsistent because:

(1) w_4 and Aw_1w_4 must be in every template.

(2) Aw_1w_5 is inconsistent in any template containing the not-arrow alternative to w_2.

(3) Therefore w_2 and Aw_1w_2 must be in every template.

(4) But Aw_1w_2 is inconsistent in any template containing the not-arrows alternative to w_3 or w_5.

Template (h) is consistent at this stage, so the next step is to build the table of forced values (Table 3.3) and the table of arrow constraints (Table 3.4). It can be deduced from the table of constraints that no consistent configuration is possible:

(a) Aw_3w_4 and Aw_5w_4 must exist by (2) and (4) and forward-connectedness.

(b) Aw_2w_5 must exist by (1) and (3) and forward-connectedness.

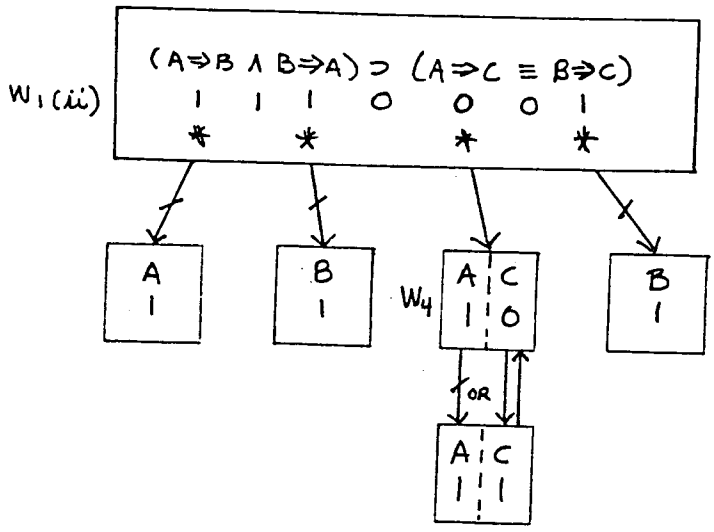
(c) Aw_2w_4 must exist by (a) and (b) and transitivity.

(d) Aw_2w_4 cannot exist by (a) and (3).

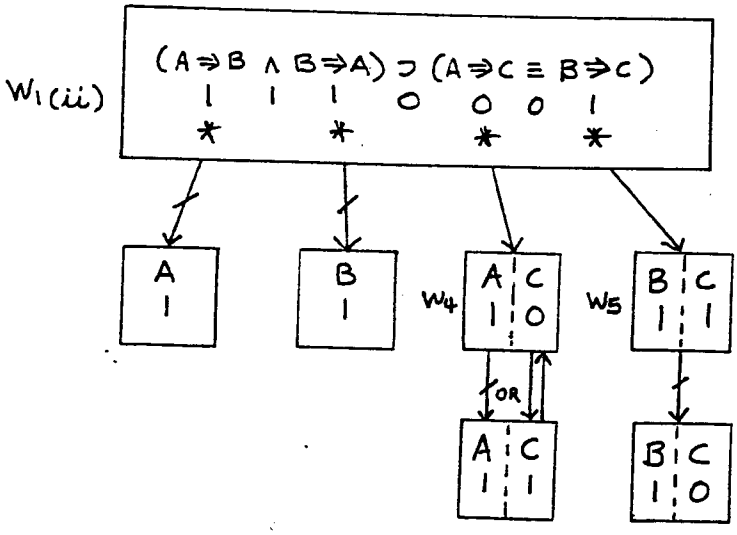
A contradiction is reached in (c) and (d). No configuration can satisfy these constraints, and the SCS is therefore inconsistent. Thus both SCS are inconsistent, and A1 is NTP-valid.

2.3. Proof of Axiom A2

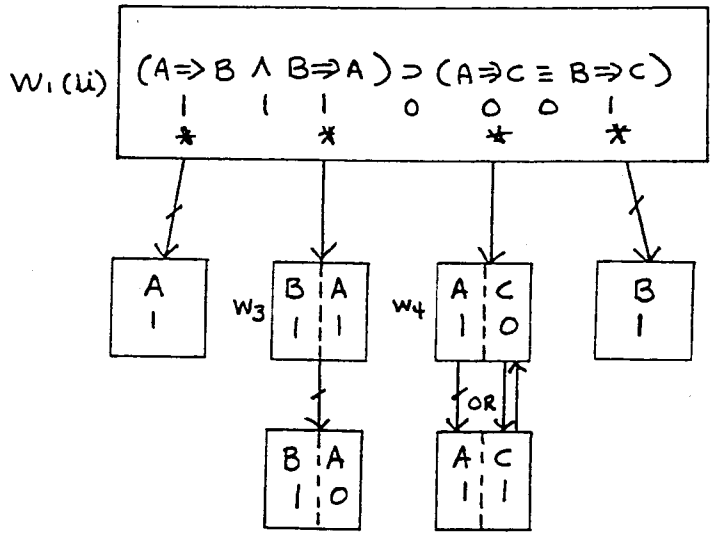
The SCS for A2 is shown in Figure 3.5. The structural templates are given in Figure 3.6. Template (a) is inconsistent because at least one of A, B must be true at rectangle w_4. Template (b) is inconsistent by the following argument. At least one of Aw_3w_4, Aw_4w_3 must be consistent by FC. Aw_4w_3 is consistent only if Aw_3w_4 coexists. But Aw_3w_4 is inconsistent since the not-arrow alternative to w_2 forces B to be assigned 1 at w_4 (and C is assigned 0 at w_4). Template (c) is determined to be inconsistent in an analogous way. Rectangles w_2 and w_4 must be connected, but Aw_4w_2 requires that Aw_2w_4 coexist. However Aw_2w_4 is inconsistent since the not-arrow alternative to w_3 forces the assignment of 1 to A at w_4 (and C is assigned 0


W1(ii)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 0 0 1
A 1
B 1
W4
A C
1 0
B 1
OR
A C
1 1

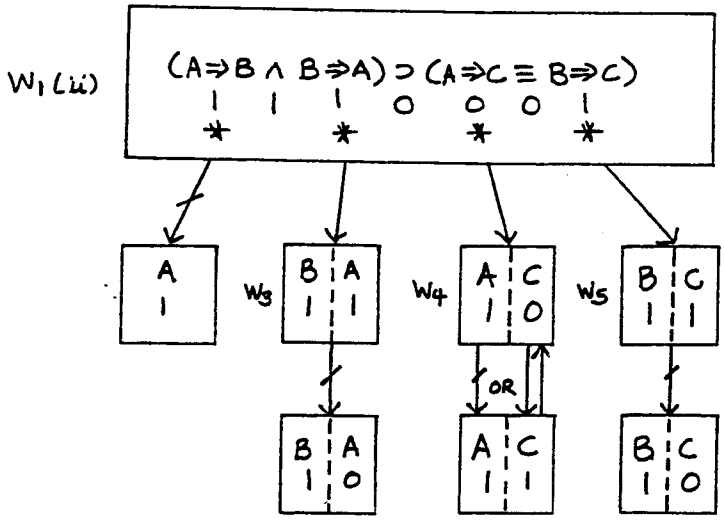
(a)


W1(ii)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 0 0 1
A 1
B 1
W4
A C
1 0
W5
B C
1 1
OR
A C
1 1
B C
1 0

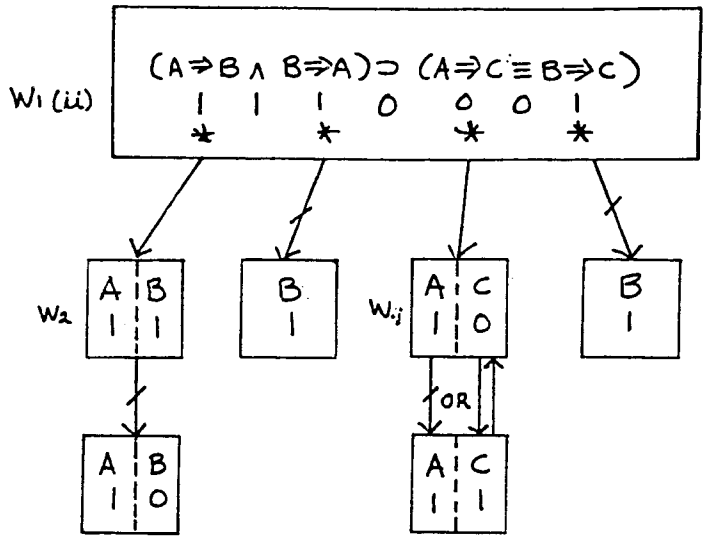
(b)


W1 (ii)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 0 0 1
A 1
W3
B A
1 1
W4
A C
1 0
B 1
OR
B A
1 0
A C
1 1

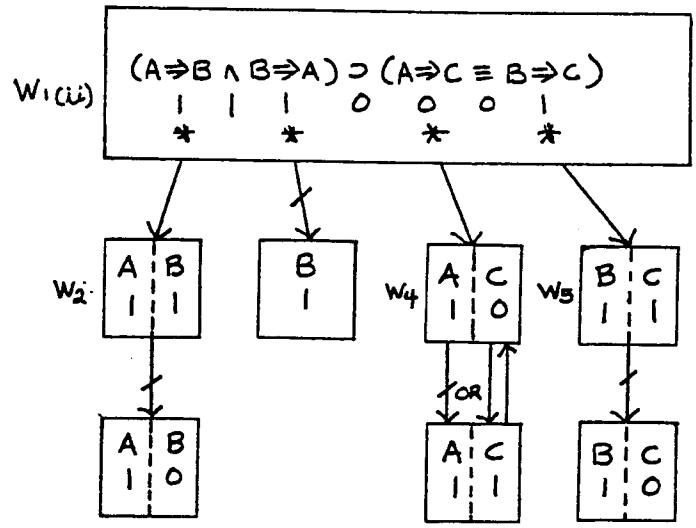
(c)


W1 (ii)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 0 0 1
A 1
W3
B A
1 1
W4
A C
1 0
W5
B C
1 1
OR
B A
1 0
A C
1 1
B C
1 0

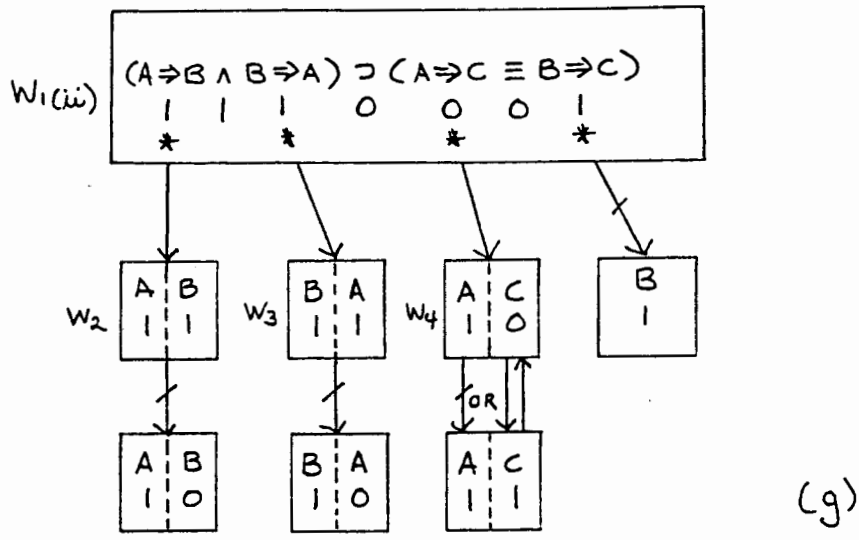
(d)


W1 (ii)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 0 0 1
* * * *
W2
A B
1 1
B
1
Wij
A C
1 0
B
1
OR
A B
1 0
A C
1 1

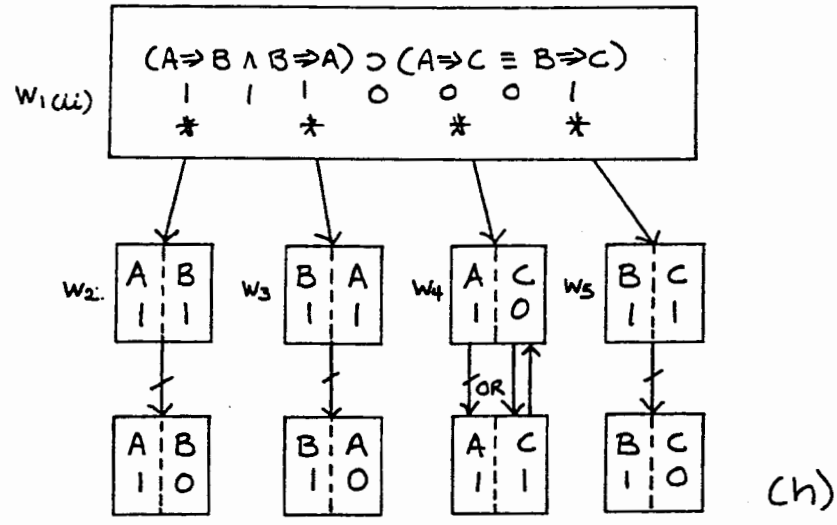
(e)


W1 (ii)
(A⇒B ∧ B⇒A) ⊃ (A⇒C ≡ B⇒C)
1 1 1 0 0 0 1
* * * *
W2
A B
1 1
B
1
W4
A C
1 0
W5
B C
1 1
OR
A B
1 0
A C
1 1
B C
1 0

(f)



(g)



(h)

Figure 3.4 Structural templates for Figure 4.2(b).

Table 3.3. Forced values for SCS(b) for A1.

Arrows	A	B	C
Aw_3w_2	1	1	
Aw_4w_2	1	1	0
Aw_5w_2	1	1	1
Aw_2w_3	1	1	
Aw_4w_3	1	1	0
Aw_5w_3	1	1	1
Aw_2w_4	1	1	0
Aw_3w_4	1		0
Aw_5w_4	1	0	0
Aw_2w_5		1	1
Aw_3w_5	1	1	1
Aw_4w_5	0	1	1

Table 3.4. Arrow constraints for SCS(b) for A1.

(1) If Aw_5w_2 exists then Aw_2w_4 and Aw_5w_4 must coexist

(2) If Aw_5w_3 exists then Aw_3w_4 and Aw_5w_4 must coexist

(3) Aw_2w_4 and Aw_5w_4 cannot coexist

(4) If Aw_3w_5 exists then Aw_5w_4 and Aw_3w_4 must coexist

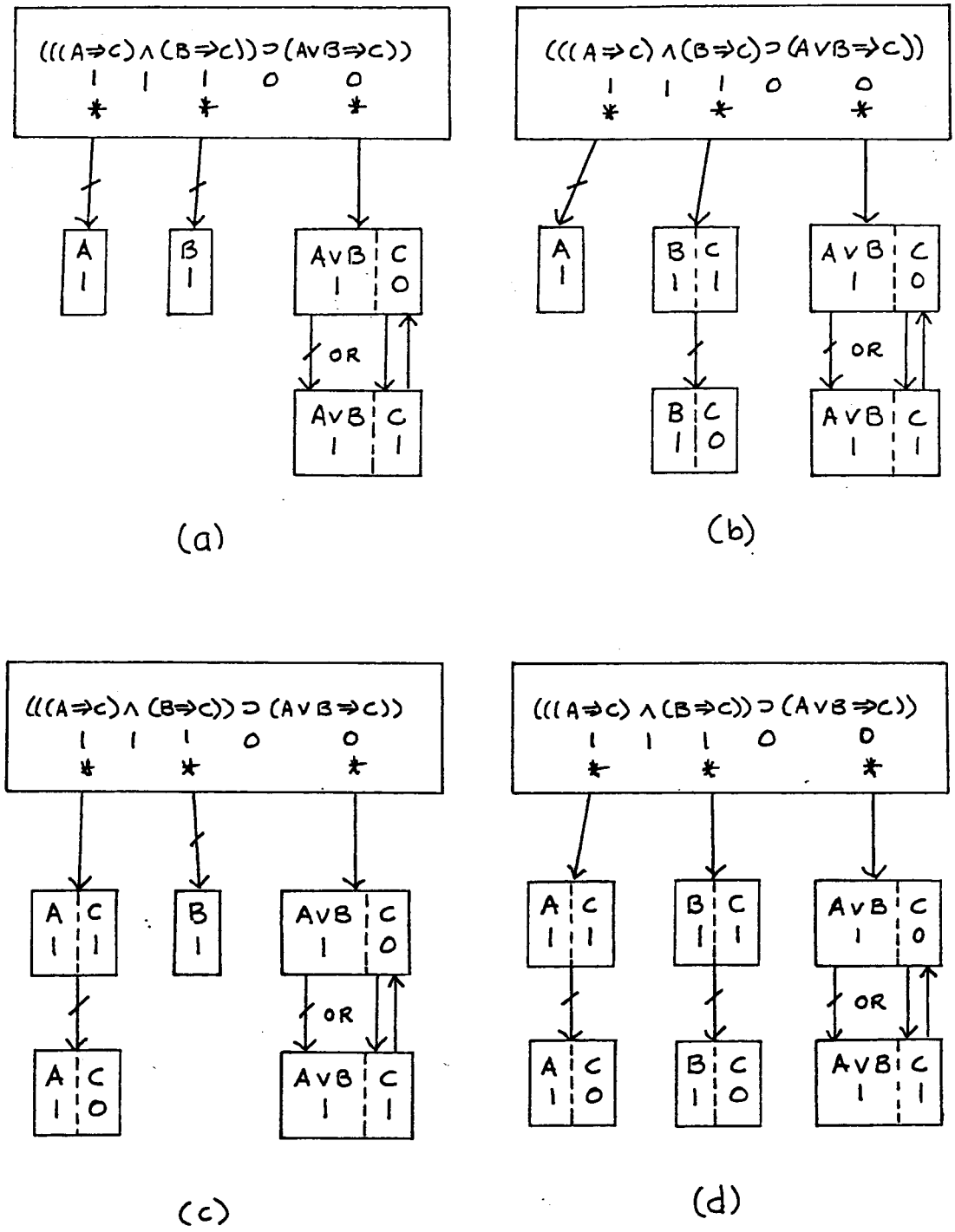


Figure 3.6 Structural templates from semi-complete structure for A2.

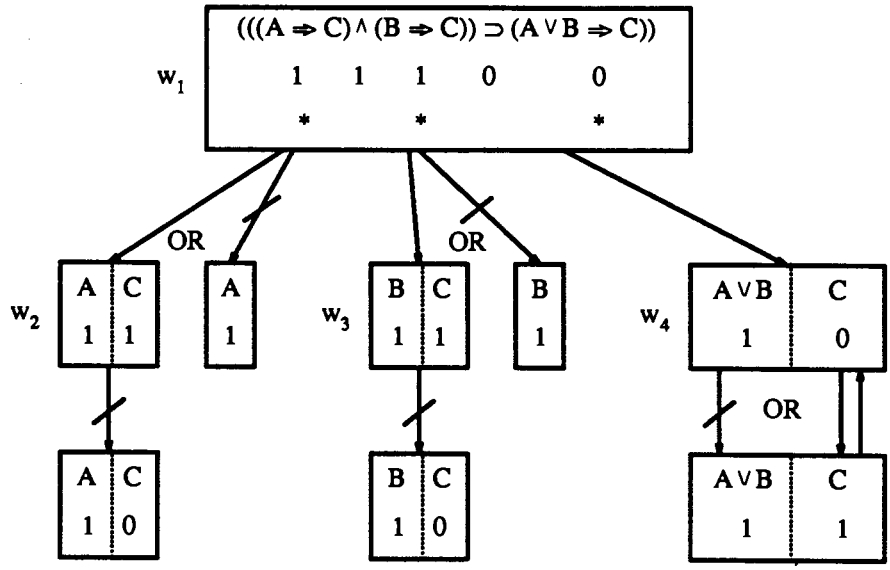


Figure 3.5. Semi-complete system of N-diagrams for $\omega = (((A \Rightarrow C) \wedge (B \Rightarrow C)) \supset (A \vee B \Rightarrow C))$.

there).

Template (d) is next considered. Table 3.5 of forced values for A2 is constructed and the derived constraints stored in Table 3.6. A contradiction is reached in (1) and (2) and (3). No configuration can satisfy these constraints, and the SCS is therefore inconsistent. It follows that A2 is NTP-valid.

2.4. Proof of Rule R0

All that remains for the soundness proof is to show that rule RO is validity preserving. R0 states that if $\vdash (B_1 \wedge \cdots \wedge B_n) \supset B$ then $\vdash (A \Rightarrow B_1 \wedge \cdots \wedge A \Rightarrow B_n) \supset A \Rightarrow B$. Thus it must be shown that if $(B_1 \wedge \cdots \wedge B_n) \supset B$ is valid, then $(A \Rightarrow B_1 \wedge \cdots \wedge A \Rightarrow B_n) \supset A \Rightarrow B$ is valid. A wff ω is valid iff it is true in all worlds in all models, i.e., there can be no world where ω is false. This condition is forced with $\neg\omega \Rightarrow \omega$, and we obtain the SCS of Figure 3.7.

Rectangle w_2 is explicitly inconsistent. Therefore every consistent configuration must include the not-arrow alternative to w_2. The not-arrows alternative to w_3, ..., w_{n+2} are inconsistent with the conditions at rectangle w_{n+3}. But rectangle w_{n+3} must occur in every consistent configuration. Thus there is only one structural template consistent at this point, the template of Figure 3.8. Labelling of the rectangles is altered

Table 3.5. Forced values for A2.

Arrows	A	B	C	A ∨ B
Aw_3w_2	1	1		
Aw_4w_2	1		1	
Aw_2w_3		1	1	
Aw_4w_3		1	1	
Aw_2w_4	0	1	0	1
Aw_3w_4	1	0	0	1

Table 3.6. Arrow constraints for A2.

(1) Aw_2w_4 must exist

(2) Aw_3w_4 must exist

(3) Aw_2w_4 and Aw_3w_4 cannot coexist

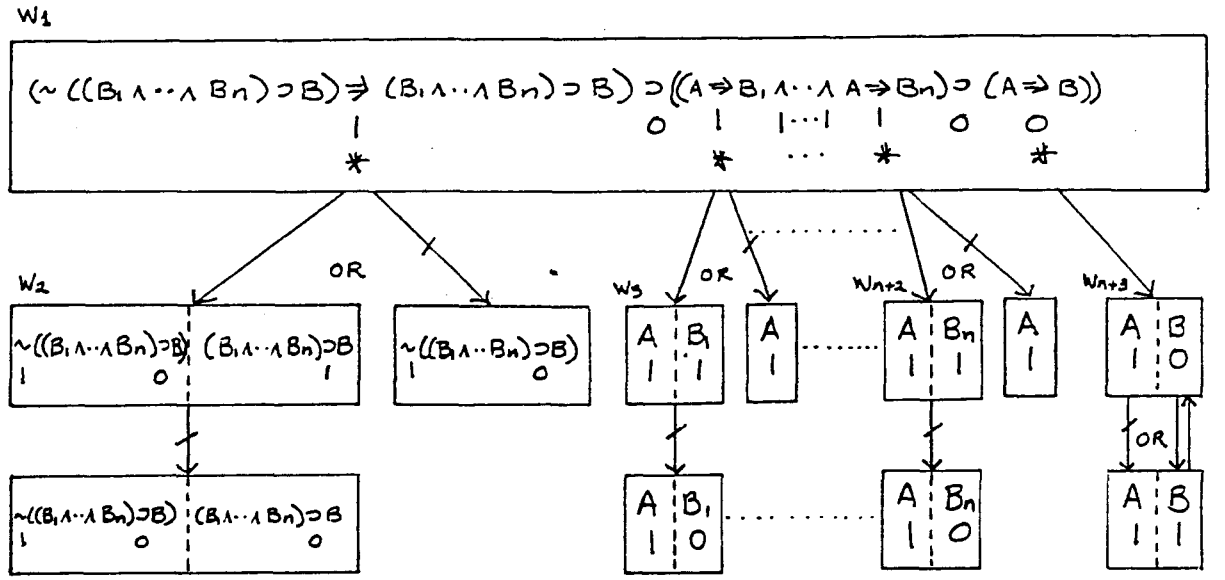


Figure 3.7 Semi-complete structure for R0.

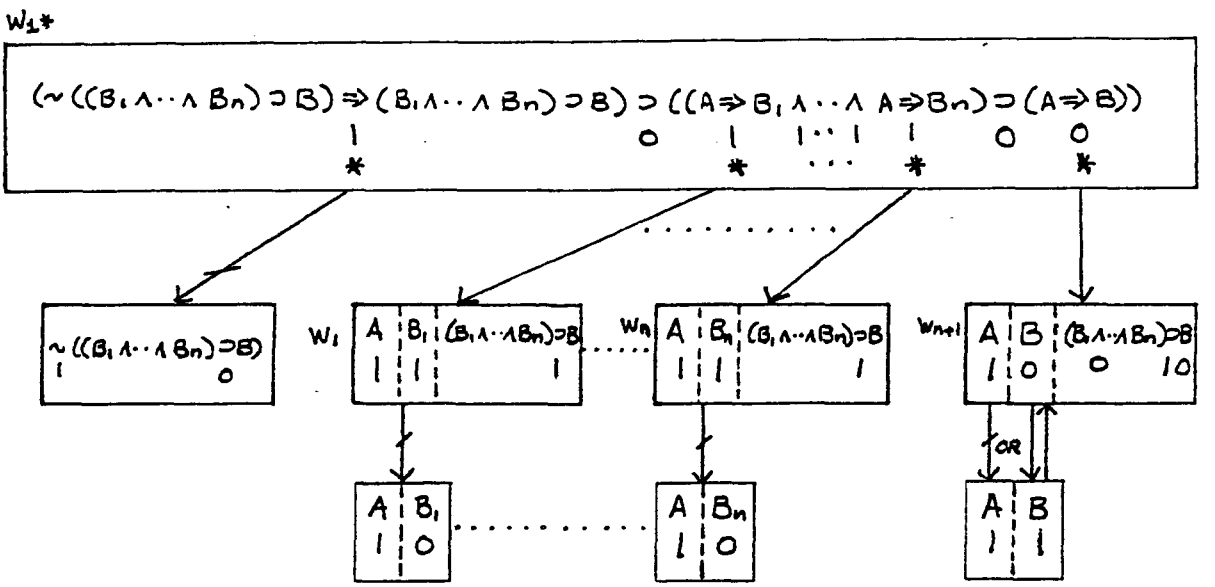


Figure 3.8 Structural template for R0.

so that the subscript of the rectangle label matches the subcript of B for $B_1, ..., B_n$. Label w_1 is changed to w_1^*.

Note that at rectangle w_{n+1} $B_1 \wedge \cdots \wedge B_n$ must be assigned 0 since B is 0 and $(B_1 \wedge \cdots \wedge B_n) \supset B$ is assigned 1. In order that $B_1 \wedge \cdots \wedge B_n$ be assigned 0 at least one of $B_1, ..., B_n$, say B_i, must be assigned 0 at w_{n+1} as in Figure 3.9. Forward-connectedness forces at least one of Aw_iw_{n+1}, $Aw_{n+1}w_i$ to exist. However, Aw_iw_{n+1} is inconsistent since A is assigned 1 and B_i 0 at w_{n+1}. Thus $Aw_{n+1}w_i$ must exist. Because Aw_iw_{n+1} is inconsistent and A is assigned 1 at w_i, B must be assigned 0 at w_i as in Figure 3.10. The same pattern arises at w_i as at w_{n+1}. B is assigned 0, and $(B_1 \wedge \cdots \wedge B_n) \supset B$ is assigned 1. Therefore at least one of $B_1, ..., B_n$, say B_j, $j \neq i$, must be assigned 0 at w_i.

One of Aw_iw_j, Aw_jw_i must be consistent, but Aw_jw_i is inconsistent since A is assigned 1 and B_j 0 at w_i. In order for Aw_iw_j to be consistent, B_i must be assigned 1 at w_j as in Figure 3.11. Transitivity requires that $Aw_{n+1}w_j$ exist. Since A is assigned 1 at w_j, either B is 0 at w_j or Aw_jw_{n+1} coexists. If Aw_jw_{n+1} exists, though, Aw_iw_{n+1} must coexist by transitivity. Aw_iw_{n+1} has already been shown to be inconsistent, however. Therefore B must be assigned 0 at w_j as in Figure 3.12. The pattern continues thusly (see Figure 3.13). It must be the case that at least one of $B_1, ..., B_n$, say B_k, $k \neq i, j$, is assigned o at w_j. Aw_jw_k must exist since Aw_kw_j is inconsistent. Transitivity forces Aw_iw_k and $Aw_{n+1}w_k$. Aw_kw_{n+1} cannot exist because Aw_iw_{n+1} has already been shown to be inconsistent.

This pattern continues for all B_x, $1 \leq x \leq n$. Rectangle w_{n-1} requires that at least one of $B_1, ..., B_n$ be assigned 0 at w_n. But only one unassigned B_x remains. The other B_x's must be assigned 1 at w_{n-1}.[9] We obtain the situation in Figure 3.14. The double-shafted dotted arrow between w_k^* and w_{n-1} represents an arrow from w_{n-2} plus all transitive arrows coming into w_{n-1}. It is used to simplify the picture.

Note that at w_n all the previous B_x are assigned 1 as well as B_n. Thus $B_1 \wedge \cdots \wedge B_n$ is assigned 1. Since $(B_1 \wedge \cdots \wedge B_n) \supset B$ must be assigned 1, B must also be assigned 1. But the transitive arrow $Aw_{n+1}w_n$ requires that Aw_nw_{n+1} coexist if A and B are both assigned 1 at w_n. Aw_nw_{n+1} cannot coexist, though. If it does, transitivity requires that Aw_iw_{n+1} coexist, and that arrow has already been demonstrated

[9] The subscript n does not necessarily match the n of $B_1, ..., B_n$. It just refers to the fact that *all* of the B_i's have been included in the argument.

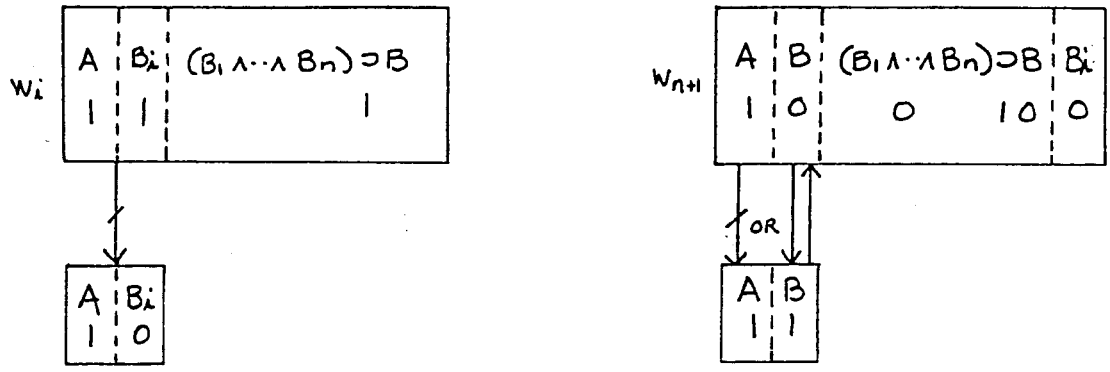


Figure 3.9 Proof of R0.

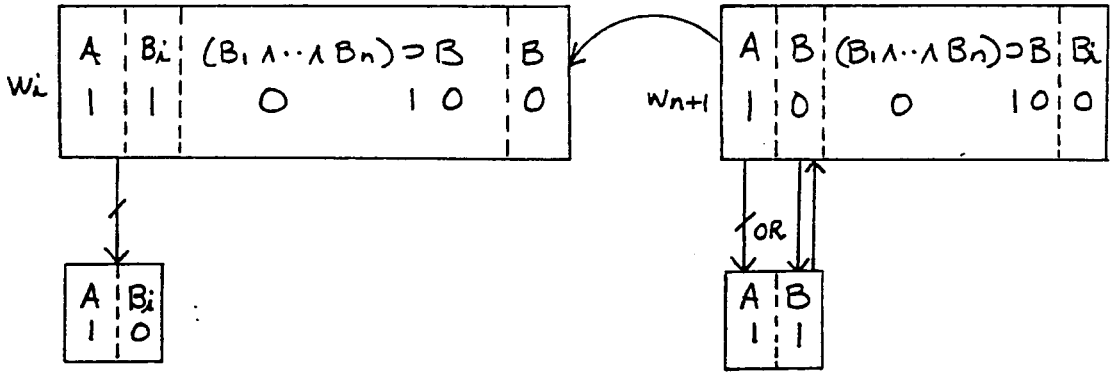


Figure 3.10 Proof of R0.

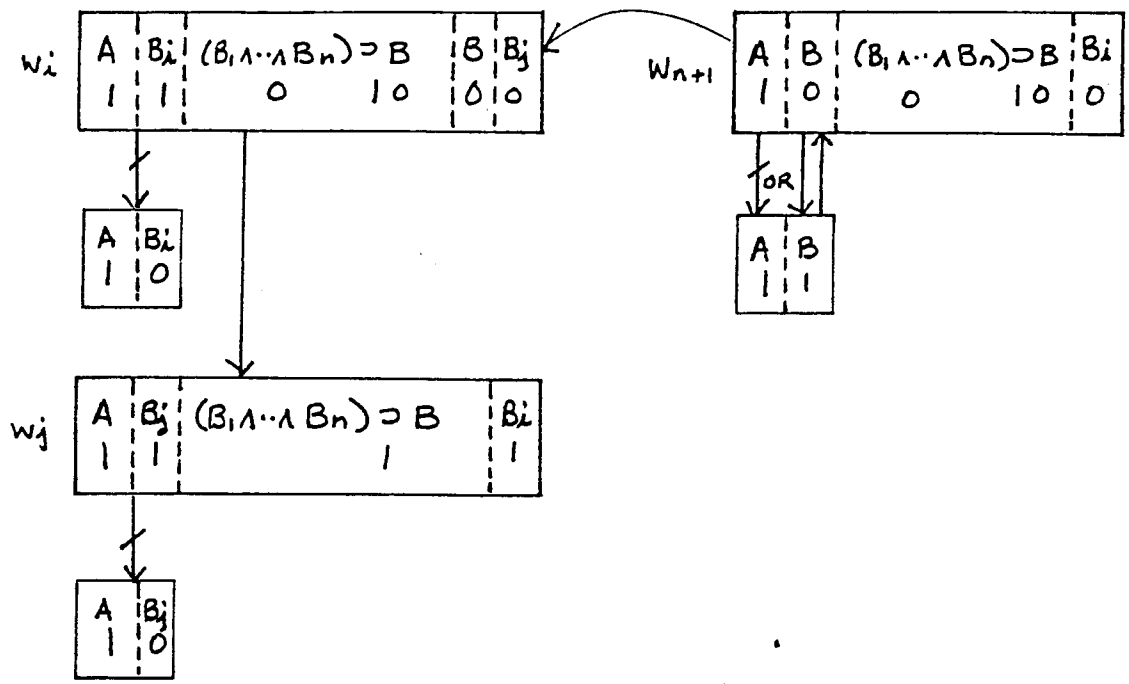


Figure 3.11 Proof of R0.

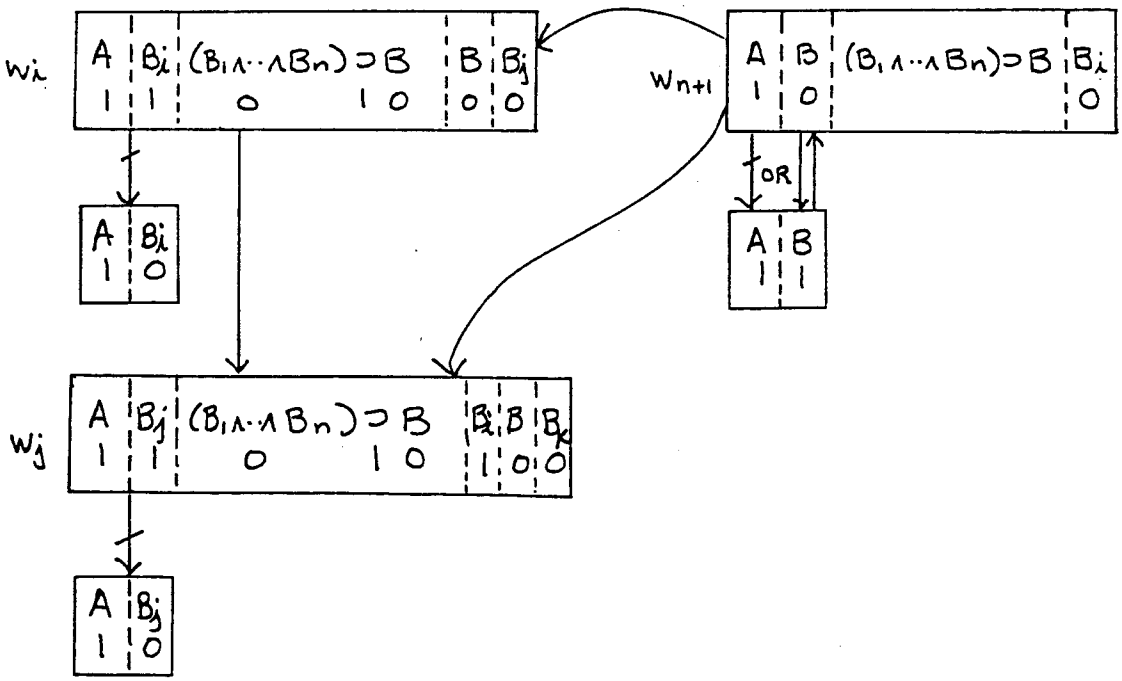


Figure 3.12 Proof of R0.

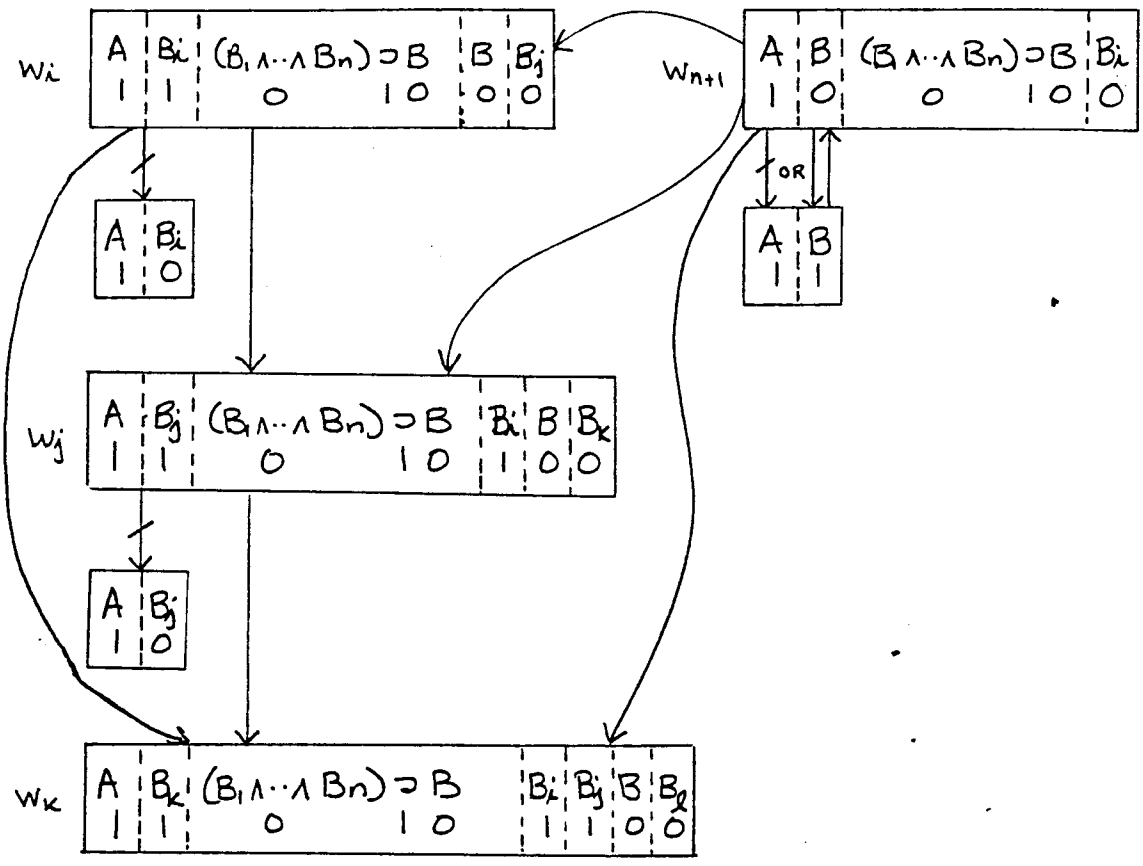


Figure 3.13 Proof of R0.

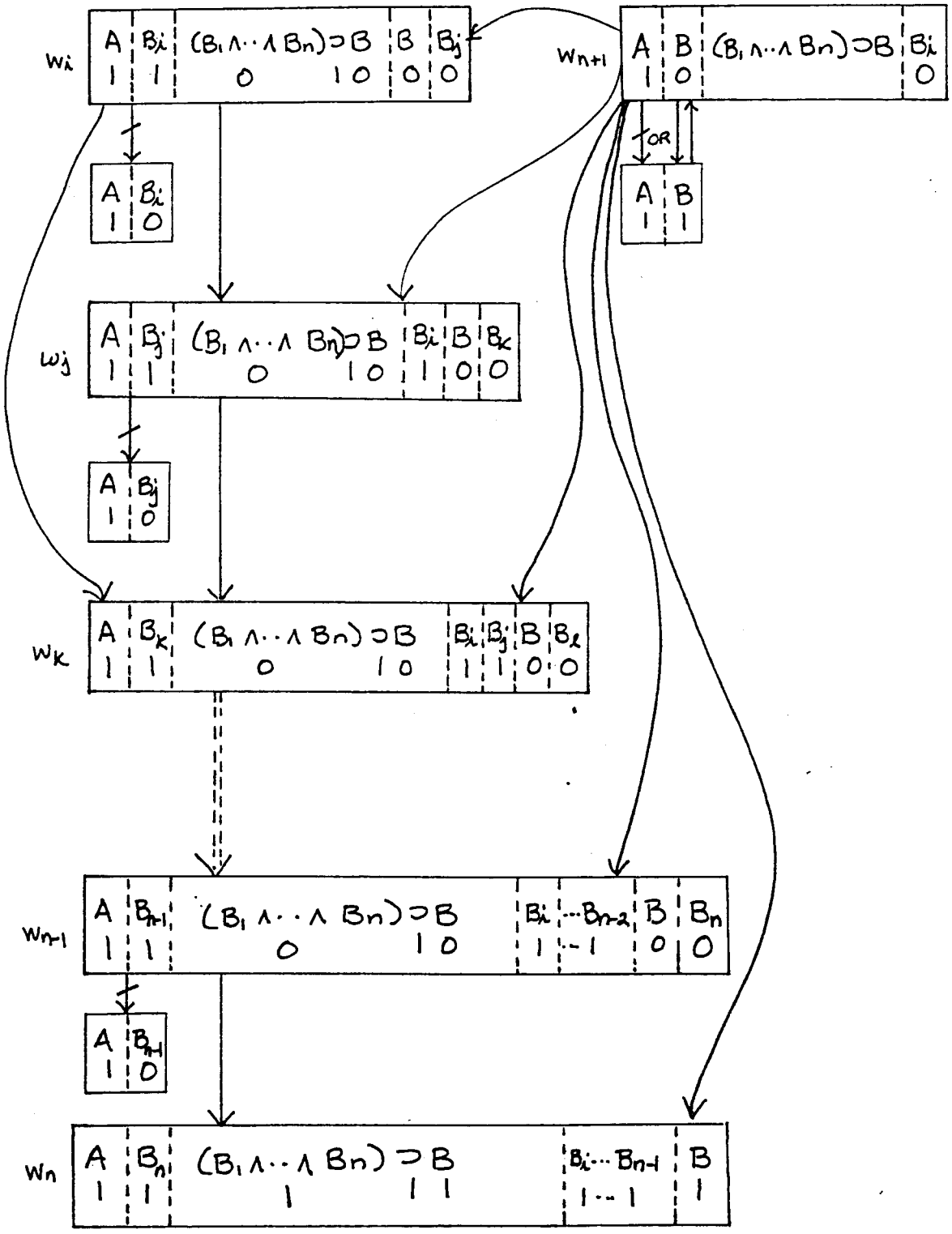


Figure 3.14 Proof of R0.

to be inconsistent. Therefore no consistent configuration is possible, and the given wff is valid.

R0 is, then, validity preserving. It has been shown that each axiom of the system is NTP-valid and that the rule preserves validity. Therefore every theorem of N is NTP-valid. It follows by transitivity from the soundness and completeness of the model theory with respect to the proof theory and lemma 3 that every valid wff is NTP-valid. This concludes the soundness proof.

3. Completeness

It remains to be proven that every NTP-valid wff is valid in the model theory.

Theorem 2. If ω is NTP-valid then $\models \omega$.

Contrapositive of theorem 2. If ω is not valid in the model theory then ω is not NTP-valid.

ω is not valid iff $\neg\omega$ is satisfiable by the definition of truth in the model theory. ω is not NTP-valid iff there exists a consistent configuration in which ω is assigned 0 at w_1. Substitution of these equivalences into the contrapositive yields:

(T2). If $\neg\omega$ is satisfiable then there exists a consistent configuration in which ω is assigned 0 at w_1.

It is T2 that is established in the following proof. Theorem 2 follows directly from T2 as shown above. Let S be any satisfiable set of formulas of N. The set S can be extended and yet retain satisfiability by lemmas 4, 5(a), 5(b).

Lemma 4. If $\alpha \in S$ and S is satisfiable, then $S \cup \{\alpha_1, \alpha_2\}$ is satisfiable.

Proof. If S is a satisfiable set containing α, an α-formula, then α is true. If α is true, then each of its operands α_1, α_2 is also true. Therefore $S \cup \{\alpha_1, \alpha_2\}$ remains satisfiable.

Lemma 5(a). If $\beta \in S$ and S is satisfiable, then at least one of $S \cup \{\beta_1\}$, $S \cup \{\beta_2\}$ is satisfiable. Alternatively:

Lemma 5(b). If $\beta \in S$ and S is satisfiable, then at least one of $S \cup \{\beta_1, \beta_2\}$, $S \cup \{\neg\beta_1, \beta_2\}$, $S \cup \{\beta_1, \neg\beta_2\}$ is satisfiable.

Proof. If S is a satisfiable set containing β, a β-formula, then β is true. If β is true, then at least one of β_1, β_2 is true. Therefore at least one of $S \cup \{\beta_1\}$, $S \cup \{\beta_2\}$ remains satisfiable. Alternatively, if β is true then

at least one of the following is true: (1) β_1 and β_2 are both true, (2) β_1 is false and β_2 is true, or (3) β_1 is true and β_2 is false. (At least one of β_1, β_2 is true in each of these possibilities.) Therefore at least one of $S \cup \{\beta_1, \beta_2\}$, $S \cup \{\neg\beta_1, \beta_2\}$, $S \cup \{\beta_1, \neg\beta_2\}$ is satisfiable.

New satisfiable sets can be spawned from S according to lemmas 6 and 7.

Lemma 6. If $\gamma \in S$ and S is satisfiable at a world w_i, then either $S_j = \{\gamma_1, \gamma_2\}$ is satisfiable at some world w_j, or $S \cup \{\neg\gamma_1\}$ is satisfiable.

Proof. If S is a satisfiable set containing γ, a $\Rightarrow$-subformula, then γ is true. If γ is true, then (1) there exists an accessible world w_j where γ_1 and γ_2 are both true, or (2) $\neg\gamma_1$ is true at all accessible worlds. Thus $S_j = \{\gamma_1, \gamma_2\}$ is satisiable at w_j, or $\{\neg\gamma_1\}$ is satisfiable at all accessible worlds. But accessibility is reflexive so that $S \cup \{\neg\gamma_1\}$ is satisfiable in case (2).

Lemma 7. If $\neg\gamma \in S$ and S is satisfiable at a world w_i, then $S_k = \{\gamma_1, \gamma_2\}$ is satisfiable at some world w_k.

Proof. If S is a satisfiable set containing $\neg\gamma$, then $\neg\gamma$ is true and γ false. If γ is false, then there exists an accessible world w_k where γ_1 is true and γ_2 false. Thus $S_k = \{\gamma_1, \neg\gamma_2\}$ is satisfiable at w_k. Q.E.D.

Let $S = \{\neg\omega\}$. S is satisfiable by the antecedent of the hypothesis. The set S can be extended in such a way that satisfiability is maintained, and new satisfiable sets spawned by applying lemmas 4, 5(a), 5(b), 6, and 7 as often as possible. If $\neg\omega$ is satisfiable then there is some world, w_1, in some model structure M in which S is satisfiable, i.e., for all S such that $p_s \in S$, $\models^M_{w_1} p_s$ ($w_1 \in P(p_s)$).

For each p_s such that p_s is a true γ, $\models^M_{w_1} \gamma_1 \Rightarrow \gamma_2$, (1) there exists a $w_j \in W$, Ew_1w_j, such that $\models^M_{w_j} \gamma_1$ and $\models^M_{w_j} \gamma_2$ and no $w_k \in W$ such that Ew_jw_k and $\models^M_{w_k} \gamma_1$ and $\models^M_{w_k} \neg\gamma_2$, or (2) $\models^M_{w_i} \neg\gamma_1$ for all wi such that Ew_1w_i. For each p_s such that p_s is a false γ, $\models^M_{w_1} \neg(\gamma_1 \Rightarrow \gamma_2)$, there exists a $w_j \in W$ such that Ew_1w_j and $\models^M_{w_j} \gamma_1$ and $\models^M_{w_j} \neg\gamma_2$ and either (1) there exists no $w_k \in W$ such that Ew_jw_k and $\models^M_{w_k} \gamma_1$ and $\models^M_{w_k} \gamma_2$, or (2) if there exists such a w_k, then Ew_jw_k.[10]

However, these are precisely the rules used to build the semi-complete structure of NTP. Propositional-rule assignments are made to subformula of the formulas within a rectangle and the truth conditions for $\Rightarrow$ used to create new rectangles. If tableaux such as Smullyan's are used at each rectangle,

[10] Substitute w_l for w_1 in the above for the nested case where $\gamma\,(\neg\gamma) \in S_l$, $l \neq 1$, i.e., $\models^M_{w_l} (\neg)\gamma_1 \Rightarrow \gamma_2$.

then lemma 5(a) applies, i.e., there is an open branch containing the elements of $S \cup \beta_1$ or $S \cup \beta_2$ if the branch containing S and $\beta \in S$ is open. If the rule for alternatives is used instead, we get three (or two) new rectangles $w_{j(i)}$, $w_{j(ii)}$, $w_{j(iii)}$ containing $S \cup \{\beta_1, \beta_2\}$, $S \cup \{\neg\beta_1, \beta_2\}$, $S \cup \{\beta_1, \neg\beta_2\}$ respectively. Rectangle w_j is inconsistent iff each of $w_{j(i)}$, $w_{j(ii)}$, $w_{j(iii)}$ is inconsistent. (Proof in [Hughes and Cresswell 68, pp. 100-101, 114-115].) Thus w_j is consistent iff at least one of $w_{j(i)}$, $w_{j(ii)}$, $w_{j(iii)}$ is consistent.

New rectangles created by the rule for new worlds contain γ_1, γ_2 assigned 1 if $\gamma_1 \Rightarrow \gamma_2$ is assigned 1 at the world from which the rule was invoked, and γ_1 assigned 1, γ_2 assigned 0 if $\gamma_1 \Rightarrow \gamma_2$ is assigned 0. Not-arrows to unlabelled rectangles indicate conditions on accessibility from associated labelled rectangles.

The accessibility relation E connects the elements of the set W in such a way that reflexivity, transitivity, and forward-connectedness hold in the model structure M, but forward-connectedness fails in the semi-complete structure. Thus it remains to be shown that (1) NTP correctly generates structural templates from an SCS, (2) NTP correctly generates RTFC arrow sets, and (3) NTP correctly performs consistency tests. If each of these conditions is true, then if $\neg\omega$ is satisfiable then NTP finds a consistent configuration. It has already been shown that the algorithm terminates in a finite number of steps and is unambiguous.

3.1. Structural Template Generation

The algorithm for generating structural templates from an SCS builds a tree in which the set of nodes along each branch forms a template. The root contains w_1 plus the set of labels for rectangles in the SCS created by the rule for a false $\Rightarrow$. This set is then included in every branch since it is at the root of the tree.

Algorithm Structural-Template

1. $root \leftarrow \{w_1\}$
2. For each w_i in SCS created by the rule for a false $\Rightarrow$ at w_k do
   $root \leftarrow root \cup \{w_i\}$
3. For each w_i in SCS created by the rule for a true $\Rightarrow$ at w_k do
   extend tree by branching at each leaf to the left with w_i and
   to the right with the not-arrow from w_k ($w_k \neg\delta$)

Steps 1 and 2 create the root of the tree. Step 3 creates two new branches at each leaf, one to the rectangle w_i and the other to the not-arrow from w_h. For example, the algorithm generates the tree of Figure 3.15 from the SCS of Figure 3.6. The templates from this tree are shown in Figure 3.8. In this way we get templates of all combinations of labels and not-arrows, each containing only one element from each alternative label-not-arrow pair.

3.2. Arrow-set Generation

The accessibility relation imposes a well-ordering on equivalence classes of rectangle labels (the set W of the model structure M). It is this view of arrow sets as a well-ordering on equivalence classes of labels which forms the basis of the arrow-set generation algorithm. The configuration template represents this well-ordering. Given any arrow set, the configuration template can be determined by the following algorithm Configuration-Template.

Algorithm Configuration-Template

1. Remove reflexive arrows from *arrowset*
2. Place w_1 in topmost box, box_1
3. Remove all arrows Aw_1w_j from *arrowset*
4. Group arrows by w_i, the rectangle Aw_iw_j leaves
5. For each w_i in at least one Aw_iw_j do
    5.1. If w_i is not already in a box then
        Put w_i in box_i
        Put box_i just below box_1
    5.2. For each Aw_iw_j do
        If $Aw_jw_i \in$ *arrowset* then
            Put w_j in box_i
            Remove Aw_jw_i from *arrowset*
        Else Put w_j in box_j below box_i
        Remove Aw_iw_j from *arrowset*
6. Add arrows from each box_i to box_j such that box_j is below box_i

Every w_j is accessible from w_1 because each w_j was created by one of the following rules:

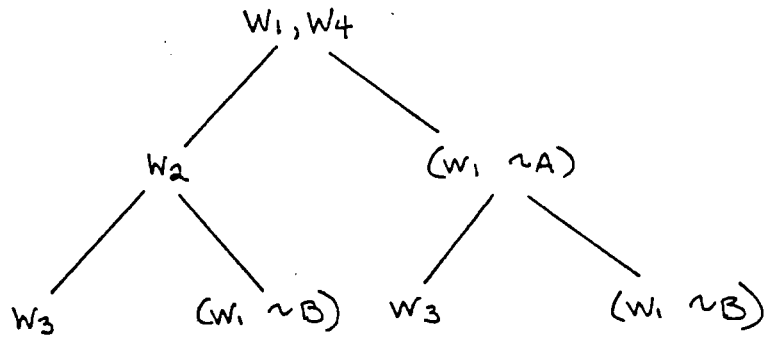


Figure 3.15 Tree of structural templates.

(1) rule for new worlds applied at w_1,

(2) rule for new worlds applied at w_i in the nested $\Rightarrow$ case, or

(3) rule for alternatives.

In case (1) Aw_1w_j is an arrow of the SCS, in case (2) Aw_1w_j is an arrow of the structural template by transitivity. In case (3) w_j is actually one of $w_{j(i)}$, $w_{j(ii)}$, $w_{j(iii)}$ created due to a β-subformula at w_j. Each arrow entering w_j then enters each of its alternatives. Thus $Aw_1w_{j(i)}$ ($Aw_1w_{j(ii)}$, $Aw_1w_{j(iii)}$) exists.

By rule FC, then, each pair of rectangles other than w_1 must be connected. w_1 is already connected by Aw_1w_j. We do not need to consider arrows Aw_jw_1 for the following reason. If Aw_1w_j or rectangle w_j is inconsistent then any configuration containing Aw_1w_j and Aw_jw_1 is still inconsistent. Conversely if a configuration containing both Aw_1w_j and Aw_jw_1 is consistent, then if Aw_jw_1 is removed the resulting configuration is still consistent.

Likewise we do not have to consider reflexive arrows when applying the rule for new worlds. For example, consider $A \Rightarrow C$ assigned 1 at w_i. Let w_i be the accessible rectangle in which A and C are both assigned 1, assuming this can consistently be done. Accessibility from w_i is now further constrained by the restriction that there can be no Aw_iw_k where A is assigned 1 and C 0 at w_k. But if this is consistent then so is a configuration containing w_j created by the rule for new worlds to make $A \Rightarrow C$ true at w_i. This is so because (1) transitivity requires that for each arrow $Awhw_i$, $Awhw_k$ must also exist since Aw_iw_j exists, and (2) forward-connectedness requires that for each arrow Aw_iw_k, w_j and w_k must be related. However, a configuration omitting w_j could be inconsistent while a configuration including w_j is consistent.

The arrow-set generation algorithm, then, must be capable of finding all possible ways of connecting all pairs of rectangles such that the properties of RTFC hold. Let W be the set of rectangle labels. Because RTFC accessibility imposes a well-ordering on equivalence classes of labels, we need to find all possible combinations of EQ classes and all possible orderings of the classes. Each combination of EQ classes $C_1, ..., C_q$ is a set of subsets of W such that each element of W occurs in exactly one C_i. A particular combination of EQ classes is represented by a configuration template. If the template contains the cardinalities of the EQ classes as opposed to the elements of W (labels), then the template is referred to as an EQ template.

Let q = the number of EQ classes in a template, n = the cardinality of W, and s_i = the cardinality of C_i. The range of s_i is from 1 to n, and therefore q also ranges from 1 to n. $s_i = n$ when $q = 1$, and $s_i = 1, 1 \leq i \leq n$, when $q = n$. Let *eqtemp* be an EQ template and *cftemp* a configuration template. Step 3 of algorithm NTP involved in generating and testing configurations is given in algorithm NTP-Step-3.

Algorithm NTP-Step-3

1. Repeat {for each SCS}
2. Generate structural templates from SCS
3. Build tables of forced values and arrow constraints
4. Repeat {for each structural template}
5. $q \leftarrow 0$
6. Repeat {for each *eqtemp* of $q = 1$ to n classes}
7. $q \leftarrow q+1$
8. Initialize *eqtemp* for q classes
9. Repeat {for each *eqtemp* of q classes}
10. Initialize *cftemp* from *eqtemp*
11. Repeat {for each *cftemp*}
12. Generate *arrowset*
13. Test configuration
14. $d \leftarrow q-1$
15. Get next *cftemp*

Until $d = 0$ {all configurations tested} or ccf

16. $c \leftarrow q-1$
17. Get next *eqtemp* of q classes

Until $c = 0$ {all *eqtemps* of q classes tested} or ccf

Until $q = n$ {all *eqtemps* tested} or ccf

Until all structural templates tested or ccf

18. Get next SCS

Until all alternative SCS processed or ccf

As soon as a consistent configuration is found, the algorithm terminates. Otherwise all possible relevant RTFC configurations are generated and tested. The repeat loop of step 6 ensures that all *eqtemps* consisting of from one to n classes are generated (or a consistent configuration found). The repeat loop of step 9 ensures that all possible cardinalities and orderings of the cardinalities of EQ classes for an *eqtemp* of q classes are generated (or a consistent configuration found). The repeat loop of step 11 ensures that all possible combinations of elements of W are generated for a given *eqtemp* (or a consistent configuration found).[11] In the worst case all RTFC configurations are generated, and the algorithm must be able to achieve this.

[11] For the remainder of the discussion the clause "or a consistent configuration found" will be omitted.

Next we need to look more closely at how the templates and arrow sets are generated. The details of step 8, initializing *eqtemp* for q classes, is given below:

1. For $i \leftarrow 1$ to $q-1$ do
   $s_i \leftarrow 1$
2. $s_q \leftarrow n-q-1$

Recall that q = the number of EQ classes in the template, s_i = the cardinality of EQ class C_i, and n = the cardinality of W. Each time q is incremented (step 7) *eqtemp* is initialized to a set of q EQ classes such that $s_i = 1$ for $C_1, ..., C_{q-1}$. s_q, the cardinality of the "bottommost" class, is initialized to $n-q-1$.

When all possible arrow sets have been generated from an *eqtemp* of q classes, step 17 gets the next *eqtemp* of q classes. The details of step 17 are given in algorithm Get-next-eqtemp.

Algorithm Get-next-eqtemp

1. $c \leftarrow q-1$
2. If $c > 0$ then
   3. Repeat
      (a) $s_c \leftarrow s_c + 1$
      (b) $sum \leftarrow 0$
      (c) $sum \leftarrow$ sum of s_1 to s_c
      (d) $sum1 \leftarrow 0$
      (e) For $i \leftarrow c+1$ to $q-1$ do
         $s_i \leftarrow 1$
         $sum1 \leftarrow sum1 + 1$
      (f) $s_q \leftarrow n - sum - sum1$
      (g) If $sum + q - c > n$ then $c \leftarrow c-1$
   Until $sum + q - c \leq n$ or $c = 0$

The topmost classes are held constant while varying the bottommost classes until all combinations have been generated. Then the next higher class is incremented, and so on, in order to get all possible combinations. The current *eqtemp* is updated by incrementing the EQ class just above the bottommost class. The cardinalities of the classes above and including C_{q-1} (C_c) are summed. The cardinalities of the classes below C_c and above C_q are reset to 1. s_q for C_q is reset to the remaining number of labels $n - sum - sum1$.

The sum of the cardinalities of the classes must be equal to n, and step 3(a) allows the sum of the s_i to go over n. Thus the sum is checked in step 3(g), and if it goes over n, c is decremented. The whole process is repeated for the next EQ class above C_{q-1}, C_{q-2}, and so on. When the cardinality of the topmost class goes over $n - q + 1$, all *eqtemps* for q classes have been generated, and c becomes 0. Therefore the loop ends. To illustrate note the successive values of *eqtemp* for $q = 3$ classes and $n = 5$ labels of Figure 3.16. When Figure 3.16(c) is updated s_2 is incremented to 4. But the sum of C_1 to C_c is 5, and each C_i including C_q must have at least one element. The sum then goes over n, so c is decremented, and s_c for the next class above C_{q-1} is incremented (Figure 3.16(e)). Figure 3.16(h) is the last *eqtemp* generated for $q = 3$ classes. The next iteration increments s_1 to 4, but $4 + 1 + 1 > (n = 5)$, and c becomes 0. Control returns to the repeat loop of step 6 of algorithm NTP-Step-3 where q is incremented. Thus the loop of step 9 generates all possible combinations of cardinalities of q EQ classes. Step 6 gets all *eqtemps* of from 1 to n classes.

Next consider the repeat loop of step 11 which gets all possible *cftemps* from any given *eqtemp*. Configuration generation is based upon an ordered list of rectangle labels. Step 10 initializes the first *cftemp*. The details of step 10 are given in algorithm Initialize-cftemp.

Algorithm Initialize-cftemp

1. *subworlds* ← *worlds*
2. For $i \leftarrow 1$ to q do
   $C_i \leftarrow ()$
   For $j \leftarrow 1$ to s_i do
      $C_i \leftarrow$ append C_i (car *worlds*)
      subworlds ← (cdr *subworlds*)

Worlds is an ordered list of the labels of the structural template, q is the number of EQ classes in *eqtemp*, C_i refers to EQ class i, and s_i is the cardinality of C_i. The algorithm puts the first s_i elements of the remaining worlds into class C_i. For example, let *worlds* = (w_1, w_2, w_3, w_4, w_5) and *eqtemp* (1 2 2). Then C_1 becomes (w_1), C_2 (w_2, w_3), and C_3 (w_4, w_5). Any given *cftemp* is updated to the next successive *cftemp* in a way analogous to the updating of *eqtemp*. The difference is that a *cftemp* contains the actual labels whereas *eqtemp* contains the cardinality of each class. Thus "incrementing" a class is much more

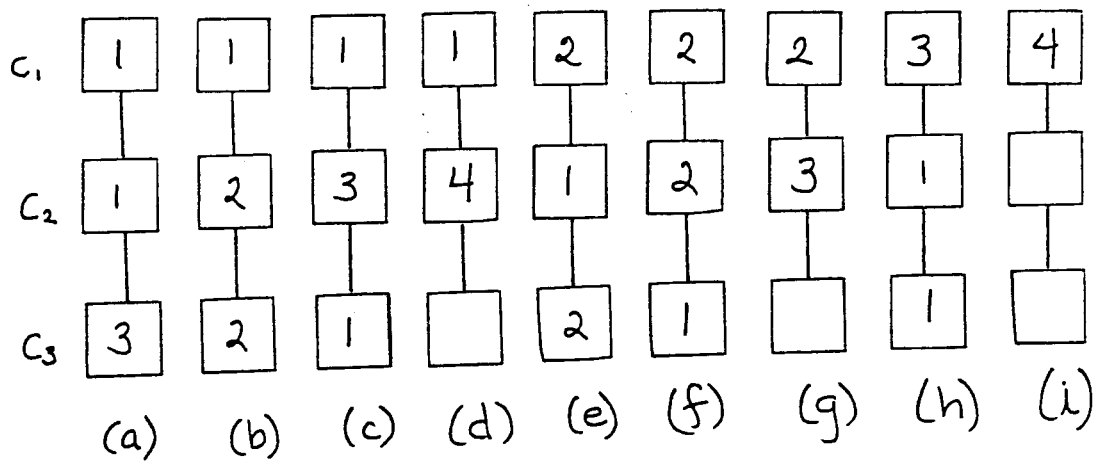


Figure 3.16 Successive equivalence templates for $q = 3$ classes and $n = 5$ world (rectangle) labels.

complicated than adding 1 to the cardinality of the class. Instead one label is removed from the class and the next successive label in *worlds* is added to the class. The details are given in algorithm Get-next-cftemp.

Algorithm Get-next-cftemp

Let *worlds* = an ordered list of the labels of the structural template
e_k = k^{th} element of C_i
q = the number of EQ classes
d = index to EQ class C
s_i = the number of worlds in C_i

1. $d \leftarrow q - 1$
2. If $d > 0$ then
   3. Repeat
      4. *subworlds* ← *worlds*
      5. For $i = 1$ to $d - 1$) do
         subworlds ← *subworlds* − C_i
      6. $k \leftarrow s_d + 1$
      7. Repeat
         $k \leftarrow k + 1$
         8. *sworlds* ← (cdr (member e_k of C_d *subworlds*))
      Until (length *sworlds*) ≥ $s_d - k + 1$ or $k = 1$
      9. If (length *sworlds*) ≥ $s_d - k + 1$ then
         10. For $i = k$ to s_d do
             $C_d \leftarrow C_d - e_i$
         11. For $i = k$ to s_d do
             $C_d \leftarrow$ append (cdr *sworlds*)
             sworlds ← (cdr *sworlds*)
         12. *subworlds* ← *subworlds* − C_d
         13. For $i = d + 1$ to q do
             $C_i = ()$
             For $j = 1$ to s_i do
                 $C_i \leftarrow$ append C_i (car *subworlds*)
                 subworlds ← (cdr *subworlds*
      Else $d \leftarrow d - 1$
   Until $d = 0$ or consistent configuration found

The symbol e_k refers to the k^{th} element of C_i, d is an index to EQ class C, and *worlds*, q, s_i are as before. Step 1 ensures that we start with $C_d = C_{q-1}$, the class just above C_q. Step 5 removes all labels in classes above C_d from *subworlds*, the set of remaining worlds. Step 8 finds e_k, the k^{th} element of C_d, (the last element of C_d in the first iteration of the loop) and sets *sworlds* to the remainder of the list of *worlds*. If *sworlds* is not long enough to fill classes C_d to C_q (repeat loop of step 7), k is decremented and the process repeated for the second to the last element of class C_d, k and so on.

If list *sworlds* is long enough to fill classes C_d to C_q, the k^{th} to the last labels of C_d are removed from C_d (step 10). These elements are then replaced with the first $s_d - K + 1$ elements of *sworlds* (step 11). The new labels of C_d are removed from *sworlds* (step 12) and the remaining classes initialized to the remaining elements of *subworlds* (step 13). If all combinations of labels within class C_d have been generated, d is decremented and the process repeated for the next class above the old C_d. When all possible combinations of labels have been tried for class C_i, d becomes 0, and the loop ends. Figure 3.13 shows the *cftemps* generated from the *eqtemps* of Figure 3.9 for worlds w_2, w_3, w_4.

Arrow sets are generated from a *cftemp* according to algorithm Arrowset.

Algorithm Arrowset

1. *arrowset* ← {}
2. Repeat
   - 2.1 For each w_i in topmost EQ class do
     - (a) For each w_j, $i \neq j$, in topmost and lower EQ classes do
       - *arrowset* ← *arrowset* ∪ $\{Aw_iw_j\}$
   - 2.2 Remove topmost EQ class

   Until no EQ classes left

A configuration template determines a unique set of arrows. Step 1 initializes *arrowset* to the empty set. Step 2.1(a) adds arrows from w_i to each w_j in the same and lower classes. The outer for loop of step 2.1 ensures that this is done for each w_i in the current topmost EQ class. Step 2.2 removes the topmost class to ensure that step 2.1 is carried out for all EQ classes in the *eqtemp*. As a result each pair of labels within any EQ class is connected by both Aw_iw_j and Aw_jw_i, and each w_i is connected to each w_j in lower EQ classes in one direction only, Aw_iw_j. Thus we get a well-ordering on EQ classes of worlds with respect to the accessibility relation E.

3.3. Consistency Testing

It has been established that the procedure is capable of finding all the relevant RTFC configurations. It must also be confirmed that only inconsistent configurations are rejected. Let w_i' be the set of relevant true sentences at rectangle w_i. A configuration is consistent if the following conditions are met:

(1) For all i, w_i' is satisfiable.

(2) If $\gamma \in w_i'$ then (a) there exists an arrow Aw_iw_j and rectangle w_j where $\gamma_1, \gamma_2 \in w_j'$ and no Aw_jw_k and rectangle w_k where $\gamma_1, \neg\gamma_2 \in w_k'$, or (b) $\neg\gamma_1 \in w_j'$ for all j such that Aw_iw_j exists.

(3) If $\neg\gamma \in w_i'$ then there exists an arrow Aw_iw_j and rectangle w_j where $\gamma_1, \neg\gamma_2 \in w_j'$ and either (a) there exists no arrow Aw_jw_k and rectangle w_k where $\gamma_1, \gamma_2 \in w_k'$, or (b) if there is such a w_k, then arrow Aw_kw_j coexists.

(4) The set of arrows in the configuration connect the rectangles in an RTFC pattern.

(5) There exists no subset of arrows $\{Aw_iw_j, ..., Aw_kw_j\}$ entering a rectangle w_j in which the conjunction of the conditions on accessibility from w_h, $i \leq h \leq k$, and the conditions at w_j are unsatisfiable.

If any one of these conditions is violated, the configuration is inconsistent. The first three conditions are concerned with the set of sentences true at a rectangle w_i. To use the terminology of Smullyan [68], a Hintikka set is defined as follows. Let α be an α-formula with operands α_1, α_2; β be a β-formula with operands β_1, β_2; γ be a true $\Rightarrow$-formula and $\neg\gamma$ a false $\Rightarrow$-formula both with operands γ_1, γ_2. A $\Rightarrow$-formula is a formula in which the main operator is a $\Rightarrow$. A Hintikka set is a set satisfying the following conditions for all α, β, γ, $\neg\gamma$ in w_i':

H_0: No variable and its negation are both elements of w_i'.

H_1: N γ and its negation $\neg\gamma$ are both elements of w_i'.

H_2: If $\alpha \in w_i'$, then $\alpha_1 \in w_i'$ and $\alpha_2 \in w_i'$.

H_3: If $\beta \in w_i'$, then $\beta_1 \in w_i'$ or $\beta_2 \in w_i'$.

H_4: If $\gamma \in w_i'$, then (1) there exists a $w_j \in W$ such that Ew_iw_j, and $\gamma_1 \in w_j'$ and $\gamma_2 \in w_j'$ and there is no $w_k \in W$ such that Ew_jw_k and $\gamma_1 \in w_k'$ and $\neg\gamma_2 \in w_k'$, or (2) there exists no w_j such that Ew_iw_j and $\gamma_1 \in w_j'$.

H_5: If $\neg\gamma \in w_i'$, then there exists a $w_j \in W$ such that Ew_iw_j and $\gamma_1 \in w_j'$ and $\neg\gamma_2 \in w_j'$ and either (1) there exists no $w_k \in W$ such that Ew_jw_k and $\gamma_1 \in w_k'$ and $\gamma_2 \in w_k'$, or (2) if there exists such a w_k, then Ew_kw_j.

Lemma 8. Every Hintikka set is satisfiable.

Proof. The analogous proofs for Hintikka sets of propositional and first-order logics are given in [Smullyan 68]. The goal is to find an interpretation in which every element of the set w_i' is true. This objective is accomplished by assigning to each propositional variable and γ-subformula which occurs in at least one element of w_i' a truth value as follows:

(1) If $p \in w_i'$, assign the value true to p.

(2) If $\neg p \in w_i'$, assign the value false to p.

(3) If neither $p \in w_i'$ nor $\neg p \in w_i'$, then assign true or false to p, say true.

(4) If $\gamma \in w_i'$, then assign the value true to γ.

(5) If $\neg\gamma \in w_i'$, then assign the value false to γ.

(6) If neither $\gamma \in w_i'$ nor $\neg\gamma \in w_i'$, then assign true or false to γ, say true.

Note that rules (1) and (2) ((4) and (5)) are compatible if w_i' is a Hintikka set because of condition H_0 (H_1). Now it must be shown that every element of w_i' is true under this interpretation. The proof is an induction on the degree of the elements of w_i'. It follows immediately from rules (1) and (2) that every propositional variable is true under this interpretation. Any element X of w_i' of degree greater than 0 is either an α-, β-, or γ-subformula. If X is a γ-subformula, X is true under this interpretation by rules (4) and (5). For X an α- or β-subformula assume all elements of degree lower than X are true. Then either:

(1) X is an α-subformula. $\alpha_1 \in w_i'$ and $\alpha_2 \in w_i'$ by H_2. But α_1 and α_2 are of degree less than X and are therefore true by the induction hypothesis. If α_1 and α_2 are both true then α (X) is true.

(2) X is a β-subformula. $\beta_1 \in w_i'$ or $\beta_2 \in w_i'$ by H_3. But β_1 (β_2) is of degree less than X and is therefore true by the induction hypothesis. If one of β_1, β_2 is true then β (X) is true. Q.E.D.

Thus if the set w_i' forms a Hintikka set, then w_i' is satisfiable by lemma 8. But this is precisely what we get (or can obtain) when a consistent configuration is found and translated to an N-model. The table of forced values contains the consequential values obtained when Aw_iw_j' is assigned 1 for each Aw_iw_j in the configuration. These are the value assignments which must hold at w_j in order that arrow Aw_iw_j be consistent. In translating to the N-model, called world-values, the relevant true sentences at each world w_j are

obtained from the table of forced values in the following way:

For each w_j do
  For each arrow Aw_iw_j do
    If w_i was created due to a false $\Rightarrow$ and Aw_jw_i coexists then
      ignore Aw_iw_j
    Else add forced values to world-values for w_j

Lemma 9. The set of relevant true sentences at each world in an N-model world-values is a Hintikka set, or can be extended to a Hintikka set by lemmas 4, 5(a), 5(b), 6, 7.

Proof. At this stage the value assignment to β_1, β_2 is determined for any β-subformula at any w_j in the table of forced values either:

(1) as a consequent of the initial assignment of 1 to Aw_iw_j', or

(2) as a result of a delayed application of the rule for alternatives.

If a propositional variable A is irrelevant at a world w_j, then the value 1 is assigned to A at w_j. The arrow consistency test assigns 1 to Aw_iw_j', the conjunction of the conditions on accessibility from w_i and the conditions at w_j, and determines assignments which follow as consequences from this initial assignment. In the second phase of testing, mutual inconsistencies in the table of forced values are determined. Restrictions prohibiting the co-occurence of mutually inconsistent arrows are stored in the table of arrow constraints. Each generated arrow set is tested for compatibility with the arrow constraints. Therefore conditions H_0, H_1 for Hintikka sets are met for any world (rectangle) w_i.

All propositional variables which occur in ω are included in the table of world values and assigned values as discussed previously. Thus either H_2 is met, or if not, w_i' can be extended to include α_1, α_2 by lemma 4. Condition H_3 holds by (1) a forced value to make an arrow consistent, (2) the rule for alternatives, or (3) an extension of w_i' by lemma 5(a) or 5(b).

Conditions H_4 and H_5 are met for the following reasons. The semi-complete structure is built precisely to meet these conditions. If an accessible world (rectangle) is inconsistent, it is rejected. If a world cannot simultaneously meet all the conditions on accessibility from the worlds to which it is accessible,

then it is rejected.

Lemma 10. If w_j was created to make $A \Rightarrow C$ true (false) at w_h and w_j' is consistent, then $A \Rightarrow C$ is true (false) at every w_i such that arrow Aw_iw_j exists.

Proof. If Aw_iw_j exists then for every Aw_jw_k, Aw_iw_k must coexist by transitivity. But Aw_jw_k is consistent iff the conditions on accessibility from w_j are consistent with the conditions at w_k. The conditions on accessibility from w_j are precisely those which make $A \Rightarrow C$ true (false) at w_h, for Aw_hw_j.

$A \Rightarrow C$ must also be true (false) at w_j, for Aw_jw_j. Suppose it is not. Then we get the situation in Figure 3.17(a), (b). If w_j was created to make $A \Rightarrow C$ true at w_h then there can be no accessible w_k where A is assigned 1 and C 0; but in order for $A \Rightarrow C$ to be false at w_j there must exist such a w_k (see Figure 3.17(a)). Arrow Aw_jw_k is inconsistent, and $A \Rightarrow C$ must be true at w_j.

If w_j was created to make $A \Rightarrow C$ false at w_h then there can be no accessible w_k where A and C are both assigned 1 unless arrow Aw_kw_j also exists. There must exist such a w_k to make $A \Rightarrow C$ true at w_j (see Figure 3.17(b)), but Aw_kw_j cannot coexist due to the conditions on accessibility from w_k. Therefore $A \Rightarrow C$ must be false at w_j.

Suppose $A \Rightarrow C$ is not true (false) at some w_i such that arrow Aw_iw_j exists. Then we get the situation in Figure 3.18(a), (b). Since Aw_iw_j and Aw_iw_k coexist, forward-connectedness requires that rectangles w_j, w_k be connected (see Figure 3.18(a)). However Aw_jw_k is inconsistent, and Aw_kw_j is inconsistent because it requires the coexistence of inconsistent arrow Aw_jw_k. Thus $A \Rightarrow C$ must be true at w_i. By the same argument, $A \Rightarrow C$ must be false at w_i in Figure 3.18(b). Q.E.D.

Arrow Aw_1w_j exists in every configuration for all w_j. Thus if $A \Rightarrow C$ is true (false) at w_j (where w_j was created to make $A \Rightarrow C$ true (false) at some w_h), then $A \Rightarrow C$ is true (false) at w_1. If for all j, w_j is consistent, then all the $\Rightarrow$-subformula are simultaneously consistent at w_1. Alternatively, if any one rectangle w_j is inconsistent or a subset of arrows entering w_j is mutually inconsistent, then at least one $\Rightarrow$-subformula is inconsistent with respect to the configuration. There may be other consistent configurations, though. Thus the consistency tests reject those configurations in which conditions H_0, H_1, H_4, H_5 are contradicted, and it has been shown that w_i' can be extended to meet conditions H_2, H_3 by lemmas 4, 5(a), 5(b), 6, 7. By lemma 8 every Hintikka set is satisfiable. Thus the sets w_i' are satisfiable.

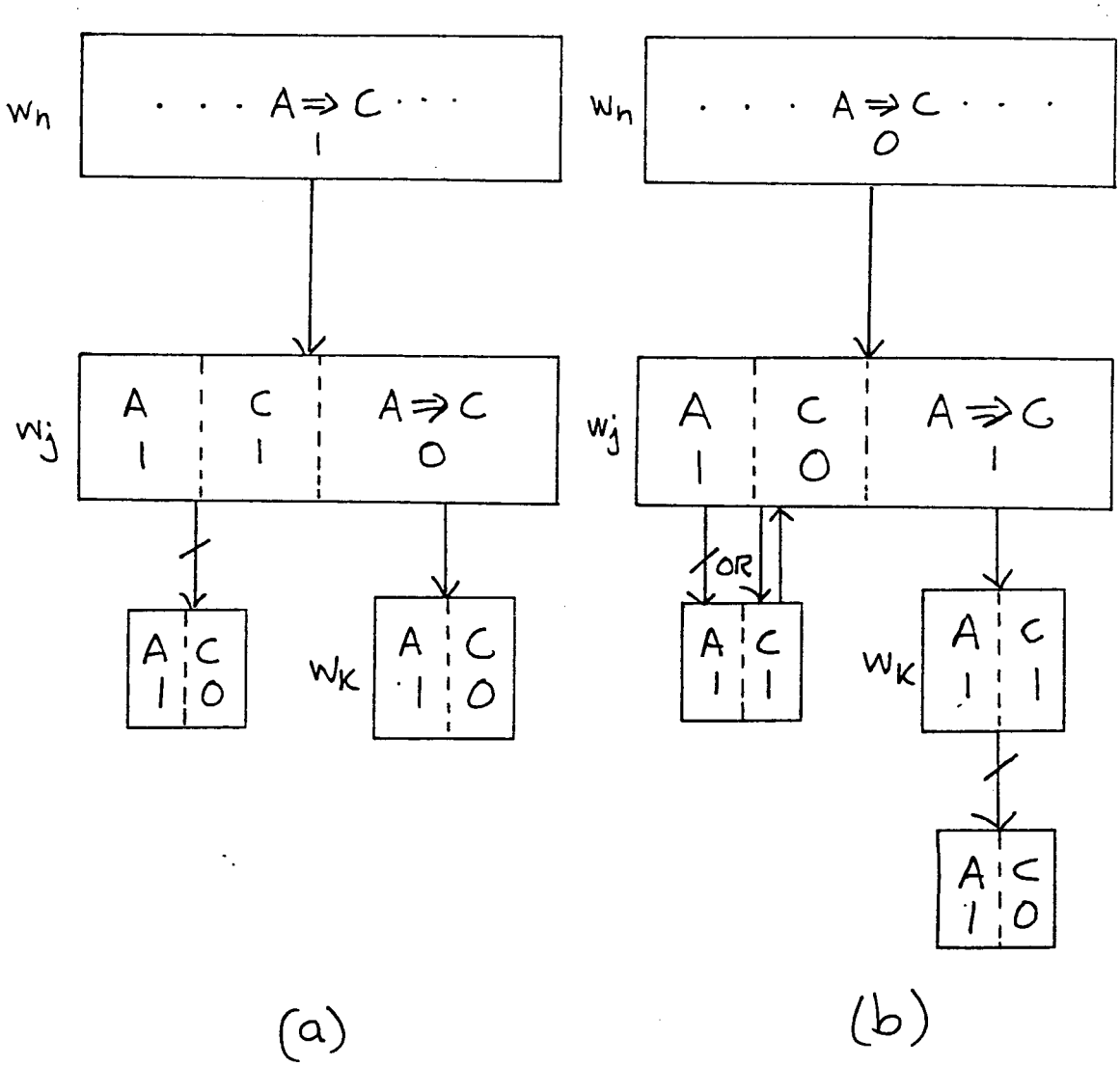


Figure 3.17 Proof of reflexive case for lemma 10.

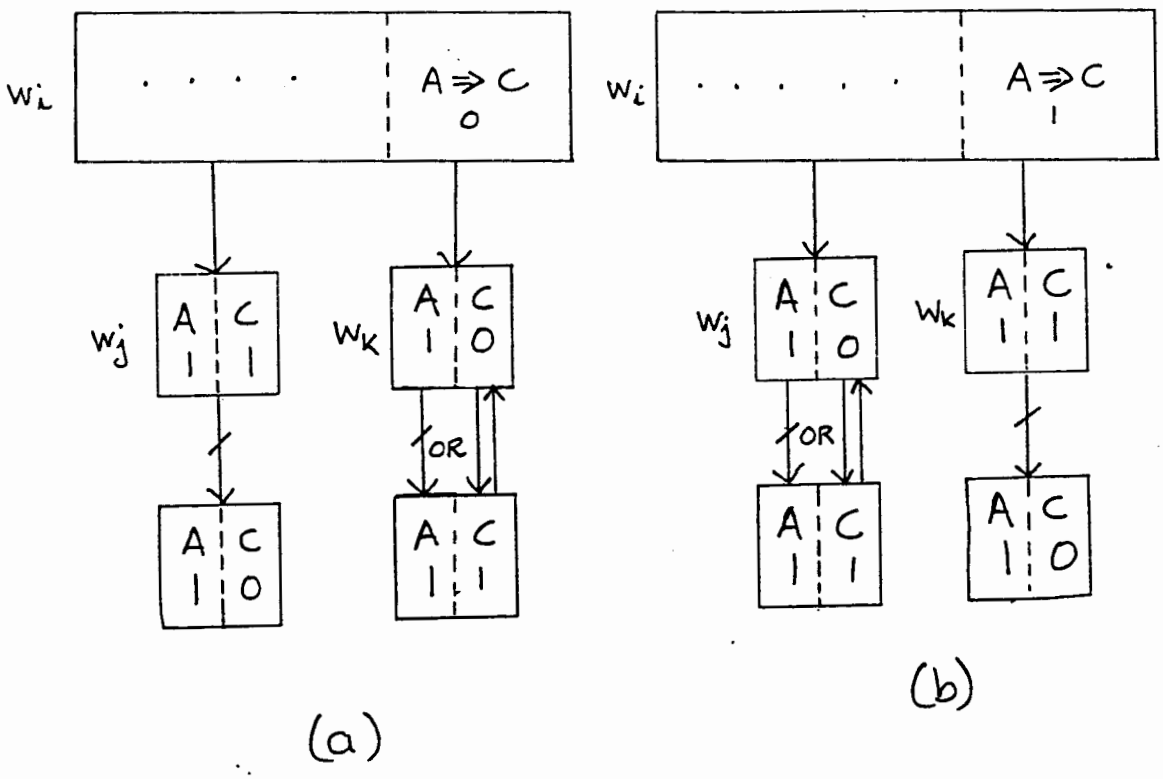


Figure 3.18 Proof of lemma 10.

Thus algorithm NTP applies rules based upon the definition of truth in the model theory in order to build a structure, SCS, consisting of the minimal elements required to find a consistent configuration for $\neg\omega$. However the SCS is missing arrows connecting all pairs of rectangles so that RTFC hold. We have seen that the algorithm is capable of producing all possible RTFC configurations. Each configuration is tested and rejected if inconsistent. If a consistent configuration is found, the configuration is translated to an N-model, the table of world values. The table of world values represents an N-model in which $\neg\omega$ is satisfiable in some world w, i.e., $\models_w^M \neg\omega$. If every configuration is inconsistent, then $\neg\omega$ is unsatisfiable and ω is valid.

The method of N-diagrams is consistent, sound, and complete with respect to the model theory of N. It has been shown in [Delgrande 87] that the model theory is sound and complete with respect to the proof theory. By transitivity, then, the method of N-diagrams is sound and complete with respect to the proof theory.

Chapter 4. An Automated Theorem Prover for N

The present chapter is concerned with the automation of the method of N-diagrams. Section 1 is an examination of the representation of a system of N-diagrams. The algorithm for the automated theorem prover is given in section 2. The main data structures are described is section 3, and an analysis of the complexity of the algorithm performance is presented in section 4.

1. The Representation

A system of N-diagrams is represented as a table consisting of three columns:

(1) world labels,

(2) conditions which hold at a world, and

(3) conditions for accessible worlds.

For example, the SCS for $\omega = (((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow (B \wedge C)))$ is shown in Figure 4.1.

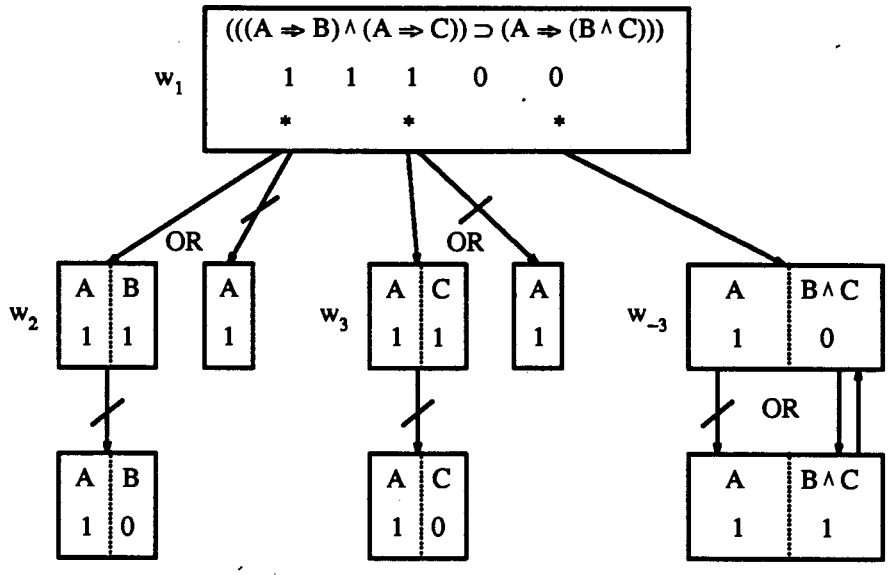


Figure 4.1. Semi-complete structure for $\omega = (((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow B \wedge C))$.

The corresponding table is shown in Table 4.1. A table representing a system of N-diagrams shall be referred to as an *N-structure*.

Table 4.1. N-structure for $\omega = (((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow (B \wedge C)))$.

worlds	conditions at world	conditions for accessible worlds
w_1	$\neg\omega$	$(Aw_1w_2 \vee \neg A) \wedge (Aw_1w_3 \vee \neg A) \wedge Aw_1w_{-4}$
w_2	$A \wedge B$	$\neg A \vee B$
w_3	$A \wedge C$	$\neg A \vee C$
w_{-4}	$A \wedge \neg(B \wedge C)$	$(\neg A \vee \neg(B \wedge C)) \vee Aw'w_{-4}$

Recall that a system of N-diagrams consists of (1) labels, (2) rectangles, (3) arrows, (4) not-arrows, (5) ORs, and (6) rules for constructing the diagrams. Labels are represented in the "world" column. The contents of labelled rectangles are represented in the "conditions at world" column. Entries in this column are the relevant conditions which are true at a given world. So $A \wedge B$ is true at world (rectangle) w_2 in Figure 4.1. The contents of unlabelled rectangles are represented in the "conditions for accessible worlds" column. Entries in this column are (1) the relevant conditions which must be true at any world accessible from the given world, or (2) arrows representing accessibility to some world. The notation Aw_iw_j is used to name an arrow from w_i to w_j. For example, the arrow from w_1 to w_{-4} in Figure 4.1 is represented in row w_1, column "conditions for accessible worlds" as Aw_1w_{-4}, i.e., w_{-4} is accessible from w_1.

The contents of unlabelled rectangles are negated and placed in the "conditions for accessible worlds column." So the contents of the unlabelled rectangle at the head of the not-arrow leaving rectangle w_2 in Figure 4.1 appears in the N-structure as $\neg A \vee B$ in row w_2, column "conditions for accessible worlds." Thus at all accessible worlds from w_2 either A is false or B is true. An OR in a system of N-diagrams is represented as a logical $\vee$ symbol in an N-structure. For example, $(Aw_1w_2 \vee \neg A)$ in row w_1, column "conditions for accessible worlds" of Table 4.1 represents the arrow from w_1 to w_2, "OR," and the contents of the unlabelled rectangle alternative to w_2. This means that either (1) there is an arrow from w_1 to w_2, or (2) $\neg A$ is true at all worlds accessible from w_1.

Note the arrow $Aw'w_{-4}$ in row w_{-4}, column "conditions for accessible worlds" of Table 4.1. w' may be thought of as a variable for a world. The notation is used to denote conditions for accessibility from a world created due to a false $\Rightarrow$. Negative numbers are used as labels for such worlds to facilitate identification of these worlds. For example, w_{-4} was created to make $A \Rightarrow (B \wedge C)$ false at w_1. At any

world w' accessible from w_{-4} $\neg A \vee \neg(B \wedge C)$ must be true or if it is false then w_{-4} must be accessible for w'.

Thus reading across the table at row w_3, for example, we get "at world w_3 A and C are both true, and at every world accessible from w_3 either A is false or C is true." The rule for new worlds is used to extend the N-structure, whereas the rule for alternatives spawns new N-structures identical to the original to which an alternative assignment is added.

2. Program Validate

The purpose of program *Validate* is to determine the validity of well-formed formulas (wffs) expressed in conditional logical system N. Since N subsumes propositional logic, the program handles wffs of propositional logic as well. A wff of N is given as input, and *Validate* returns either a message stating that the given wff is valid or a message stating that the given wff is invalid. If invalid, a set of value assignments to variables is also returned in which the wff evaluates to false.

The automated theorem prover for N is an implementation of the method of N-diagrams. An attempt is made to construct a consistent configuration for $\neg\omega$ in order to prove ω. If such a configuration can be constructed, the given wff is invalid. Otherwise such a configuration is shown to be impossible to construct, and the wff is therefore valid. The program is written in *Franz Lisp* code.

Program *Validate* is an implementation of algorithm N-Theorem-Prover. To summarize the procedure, steps 1 and 2 initialize and build a table called *nstruct* which represents the semi-complete structure of N-diagrams. Step 3 generates and tests RTFC configurations from each structural template of the semi-complete structure. Step 3 is repeated until all RTFC configurations have been generated and tested or a consistent configuration for $\neg\omega$ is found. A more detailed look at how these goals are accomplished is provided in the following discussion.

Step 1. A wff is entered in one of two ways: (1) interactively within the loop instigated by function interact, or (2) as an argument to function test. Function test calls prefix to convert *wff* to prefix notation. The main operator is assigned 0 when test calls apply-rules sending it arguments *wff* and 0. *Nstruct* is initialized to w_1 where $\neg\omega$ is true.

Algorithm N-Theorem-Prover

Input: wff
Output: valid, invalid
1. Initialize *nstruct*
    1.1. Get *wff*
    1.2. Put *wff* in prefix form
    1.3. Assign 0 to the main operator of *wff*
2. Build *nstruct*, the semi-complete structure(s)
    2.1. Repeat
        Apply α-rules
        Apply rules for crosses, asterisks, and modified rule for alternatives
        Apply rule for new worlds
    Until rules applied as often as possible
3. Repeat {for each semi-complete structure}
    3.1 Generate structural templates from SCS
    3.2. Test templates
    3.3. If template is inconsistent then remove template from *paths*
    3.4. If *paths* ≠ {} then
        3.5 Build *necvals* and *arrows* {tables of forced values and arrow constraints}
        3.6 Repeat {for each structural template}
            $q \leftarrow 0$
            3.7. Repeat {for $q = 1$ to n classes in *eqtemp*}
                $q \leftarrow q + 1$
                Initialize *eqtemp* for q classes
                3.8. Repeat {for each *eqtemp* of q classes}
                    Initialize *config* template from *eqtemp*
                    3.9 Repeat {for each *config* template}
                        Generate *arwset*
                        Test configuration
                        $d \leftarrow q - 1$
                        Get next *config* template
                    Until $d = 0$ {all *config* templates tested} or ccf
                    $c \leftarrow q - 1$
                    Get next *eqtemp* of q classes
                Until $c = 0$ {all *eqtemps* of q classes tested }or ccf
            Until $q = n$ {all *eqtemps* tested} or ccf
        Until all structural templates tested or ccf
        Get next SCS
    Until all alternative SCS tested or ccf

Step 2. Function apply-rules is called from test to apply propositional rules to *wff*. The application of the rules results in the construction of *valassns*, *mustbe*, and *gen*, data structures from which alternative value assignments to variables and gamma-subformulas are generated. Application of the rule for new worlds results in the extension of *nstruct* to the SCS once the rules have been applied as often as possible.

Step 3. The second phase of the algorithm generates and test RTFC configurations. If at any point in step 3 a consistent configuration is found, the algorithm terminates. The outer loop is repeated for each SCS as there may be more than one. Structural templates are generated from the current SCS and stored in

paths (step 3.1). The templates are tested and inconsistent templates removed from *paths* (steps 3.2, 3.3). If at least one template remains in *paths*, then *necvals*, the table of forced values, and *arrows*, the table of arrow constraints, are built (step 3.5).

The loop of step 3.6 is repeated for each structural template remaining in *paths*. Structural templates consisting of more than two worlds are not forward-connected. Therefore an arrow in one direction or the other or both must be added between each pair of worlds in the configuration to make the configuration RTFC. However, all possible ways of adding arrows need not be tested. Transitivity fails for some combinations. Thus only sets of arrows which create RTFC configurations are generated. This type of generation is accomplished in the following way.

The accessibility relations and worlds in a configuration are viewed in terms of a well-ordering on equivalence classes of worlds (steps 3.6-3.9). Reflexivity and transitivity hold, but symmetry is not imposed by the accessibility relation. Thus there may occur from one to n equivalence classes, where n is the number of worlds in the configuration, excluding w_1 (step 3.7). *Eqtemp* is the data structure representing the current equivalence class structure. The ordering of EQ classes also varies (step 3.8). For example, if there are three worlds in a configuration (excluding w_1), then the successive values of *eqtemp* are: (3), (1 2), (2 1), (1 1 1). From each *eqtemp* consisting of more than one equivalence class, there are different ways in which the world labels may be fitted into the equivalence classes (step 3.9). Consider worlds w_2, w_3, w_4 and *eqtemp* (1 1 1). There are six ways of fitting three worlds labels into three equivalence classes: (2 3 4), (2 4 3), (3 2 4), (3 4 2), (4 2 3), (4 3 2). *Config* is the data structure representing the current permutation of world labels from *eqtemp*. *Arwset* is the data structure representing the set of arrows to be added to a structural template to form an RTFC config. It is generated from *config* and tested for consistency. If consistent, a false model has been found. *Arwset*, *path*, and *wrldvals*, the set of value assignments to relevant variables and gamma-subformulas at each world in the configuration, are returned. These data structures describe the false model. If inconsistent, the next RTFC configuration is generated and tested. This cycle continues until all RTFC configurations have been tested or a consistent configuration found.

The functions of program *Validate* are of three main types: (1) those which build or modify the data structures, (2) those which perform consistency tests, and (3) those which manipulate lists. See Appendix

2 for full documentation of the automated theorem prover. The user manual is given in Part I of the appendix and the maintenance manual in Part II.

3. Data Structures

The present section is comprised of (1) descriptions of the main data structures of program *Validate* and (2) lists of the functions which manipulate them. The documentation for each data structure consists of four parts:

(1) a header comprised of the name of the data structure in bold,

(2) a short description of the structure,

(3) an example, and

(4) a list of the functions responsible for initializing, building, or modifying the data structure.

The data structures appear in the order in which they arise within the program.

wff

The formula to be tested for N-validity is entered in infix notation in one of two ways. It is either entered at the terminal and read through functions interact and readexpr or entered as an argument to function test. Functions prefix and prefixw convert *wff* to prefix notation, whereas function inorder converts a wff in prefix form to infix form. A wff in infix form is in the order *operand-operator-operand*, or *operator-operand* if the operation is negation. A wff in prefix form is in the order *operator-left operand-right operand*, or *operator-operand* if the operation is negation. For example, given $(((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow (B \wedge C)))$ in infix notation, its prefix form is $(\supset (\wedge (\Rightarrow A\, B)(\Rightarrow A\, C))(\Rightarrow A\, (\wedge B\, C)))$.

valassns, mustbe

Function apply-rules assigns 0 to the main operator of *wff* and applies the rules of propositional logic for the assignment of truth values to variables and to γ-subformula. *Mustbe* is the data structure containing top level α-assignments to variables and γ-subformula, whereas *valassns* contains a list of alternative value assignments as in the case of a true logical ∨. For example, from *wff* $(((A \Rightarrow B) \wedge (B \Rightarrow A)) \supset (A \equiv B))$ we get:

mustbe: ((⇒ A B 1) (⇒ B A 1))

valassns: (((A 1) (B 0)) ((A 0) (B 1))).

The functions which build or modify *valassns* and *mustbe* are apply-rules, apply-betarule, avalues, bvalues, alpha, beta, gamma, init-mustbe.

gen

Gen is the generator by which the next alternative set of value assignments can be obtained from *valassns* and *mustbe*. The generator consists of a list of numbers, 0-2, the length of which is equal to the maximum number of ORs in *valassns*. For example, given:

mustbe: ((A 0))

valassns: (((B 1) (C 0)) ((B 0) (C 1)) ((B 0) (C 0))),

gen is initialized to (−1). When *gen* is (0) the first set of value assignments is ((*A* 0) (*B* 1) (*C* 0)); for *gen* (1) the set of value assignments is ((*A* 0) (*B* 0) (*C* 1)); for *gen* (2) ((*A* 0) (*B* 0) (*C* 0)). The functions which build or modify *gen* are init-gen, depth, maxnum, buildgen, update-gen, next-assn.

nstruct

Nstruct is the main data structure in the program. It is a table representing a system of N-diagrams. Given the system of Figure 4.1 *nstruct* is:

```
((1  (~ wff) ( ∧ ( ∧ (∨(A 1 2) (~ A)) (∨(A 1 3) (~ A))) (A 1 -4)))
 (2  ( ∧ A B) (∨(~ A) B))
 (3  ( ∧ A C) (∨(~ A) C))
 (-4 ( ∧ A (~ ( ∧ B C))) (∨(∨(~ A) (~ ( ∧ B C))) (A wprime -4))))
```

The numbers in the first column of the table are world labels. Note that a world created from a false ⇒ is given a negative label. This makes it easy to distinguish these worlds from worlds created due to a true ⇒ when it is necessary to do so (e.g., in building arrow constraints). The second column consists of the relevant true conditions at a world. This is the information inside the rectangles of the diagrams. So for example, A and B are both true at world w_2 above.

The third column consists of the conditions which must hold at worlds accessible to the current world. Consider row 1 of the example *nstruct*. The third column contains arrows (A 1 2), (A 1 3), (A 1 -4). The conditions on accessibility from world w_1 are that:

(1) either there is an accessible world w_2 or A is false at all accessible worlds,

(2) either there is an accessible world w_3 or A is false at all accessible worlds, and

(3) there must be an accessible world w_{-4}.

At worlds accessible to world w_3 it must be the case that either A is false or C is true. (It cannot be the case that A is true and C false.) At worlds accessible to world w_{-4} (created from a false ⇒ at w_1) either (1) A is false or $(B \wedge C)$ is true (it cannot be the case that A and $(B \wedge C)$ are both true), or (2) if A and $(B \wedge C)$ are both true, then there must be an arrow back to world w_{-4}. The functions which build or modify *nstruct* are init-nstruct, build, gamma-true, gamma-false, modifiedt, modifiedf, newrowt, newrowf, update-nstruct, update-worlds.

arrows

Arrows is a data structure containing information about transitive arrows, inconsistent arrows, arrows which cannot co-occur, arrows which must co-occur, nested-arrows, and betas in arrows and/or worlds. An example structure *Arrows* follows:

```
((+ (A 2 4))
(~ (A 2 3))
(~ (A 4 2) (A 3 2))
(-> ((A -5 2) (A 3 2)) (A 2 -5))
(-> ((A -5 3) (A -6 3)) (v (A 3 -5) (A 3 -6)))
(@ (A 2 4) ((A 1 4)))
(* -6)
(* (A 4 -5))
(* (1 (~ ( ^ A B)))) )
```

Row (+ (*A* 2 4)) means that arrow (*A* 2 4) must occur in any configuration in which worlds w_2 and w_4 are both present. Row (~ (*A* 2 3)) indicates that arrow (*A* 2 3) is inconsistent and therefore cannot occur in any consistent configuration. Row (~ (*A* 4 2) (*A* 3 2)) indicates that arrows (*A* 4 2) and (*A* 3 2) cannot co-occur in any consistent configuration. Row (→ ((*A* –5 2) (*A* 3 2)) (*A* 2 –5)) means that if arrows (*A* –5 2) and (*A* 3 2) co-occur, then (*A* 2 –5) must also occur. This case arises when arrow (*A* 3 2) forces value assignments at world w_2 which in turn makes (*A* –5 2) inconsistent unless (*A* 2 –5) also occurs. Similarly row (→ ((*A* –5 3) (*A* –6 3)) (v (*A* 3 –5) (*A* 3 –6))) indicates that if arrows (*A* –5 3) and (*A* –6 3) co-occur, then either (*A* 3 –5) or (*A* 3 –6) must also occur in the configuration. Row (@ (*A* 2 4) ((*A* 1 4))) represents transitive arrows. If arrow (*A* 2 4) occurs in a configuration then arrow (*A* 1 4) must also occur since accessibility is transitive.

An asterisk indicates the occurrence of a beta-subformula. Row (* –6) means that there exists a beta-subformula within the conditions at world w_{-6}. Row (* (*A* 4 –5)) means that a beta-subformula occurs within either the conditions for accessibility from w_4 or the conditions at w_{-5}. Thus the arrow of *necvals* (*A* 4 –5) contains a beta-subformula. Row (* (1 (~ (^ *A B*)))) indicates that not-arrow (1 (~ (^ *A B*))) contains a beta-subformula. The functions which build or modify *arrows* are updated-arws, wkworlds, wkworld, gsum, update-arws, trans, get-trans, modfarws, update-arrows, double-arw, ifwiwj, ifwi, updated-arrows, cant-co-occur, cannot, cant, check-pairs, upd-arw, betanots, add-arws, beta-nestednecs.

necvals

Necvals is a data structure consisting of the set of arrows to be added to a configuration to make it FCT. Associated with each arrow is (1) the set of value assignments which must hold in order that the arrow be consistent, and (2) the term "cons" or "incons" to indicate the consistency of the arrow, for

example:

(((A 3 2) (A 1) (B 1) (C 1) cons)
((A -4 2) (A 1) (B 1) (C 0) cons)
((A 2 3) (A 1) (B 1) (C 1) cons)
((A -4 3) (A) (B) (C) cons (A 3 -4))
((A 2 -4) (A) (B) (C) incons)
((A 3 -4) (A 1) (B 0) (C 0) cons)).

From the example *necvals* we see that arrows $(A\ 3\ 2)$, $(A\ {-4}\ 2)$, $(A\ 2\ 3)$, and $(A\ 3\ {-4})$ are all consistent. Arrow $(A\ {-4}\ 3)$ is consistent as long as arrow $(A\ 3\ {-4})$ also occurs in any configuration in which $(A\ {-4}\ 3)$ occurs. Arrow $(A\ 2\ {-4})$ is inconsistent. This information is used to build *arrows*. The functions which build or modify *necvals* are test-arrows, update-necvals, check-wprime, add-vals, add-val, rtnarws, awjwi, init-necvals, generate-test.

paths

Paths is a data structure which represents all possible structural templates along the semi-complete structure of N-diagrams. Consider the system of N-diagrams of Figure 4.1. *Paths* for this semi-complete structure is:

(((A 1 -4) (1 (~ A)) (1 (~ A)))
((A 1 -4) (1 (~ A)) (A 1 3))
((A 1 -4) (A 1 2) (1 (~ A)))
((A 1 -4) (A 1 2) (A 1 3))).

The first path represents a configuration in which two worlds exist, w_1 and w_{-4}. There are two not-arrows both indicating that A must be false at all worlds accessible from w_1. The last path represents a semi-complete structure in which four worlds exist, w_1, w_2, w_3, and w_{-4}. There are no not-arrows on this path. The functions which build or modify *paths* are init-paths, transitive, intrans, complete, comp, newnrows, nested-paths, expanded, newrows, test-template, test-temp, add1base2.

worldvals, wrldvals

Data structure *worldvals* contains value assignments to variables and gamma-subformula which must hold at each world. It is initialized to the conditions which hold at each world from *nstruct*. Consider an example:

((1 (A 1) (B 1) (C 0) (⇒ A B 1) (⇒ C B 0))
(-3 (C 1) (B 0) (A 0))
(2 (A 1) (B 1) (C 0))).

At world w_1 A and B are true, C false, $A \Rightarrow B$ true, and $C \Rightarrow B$ false. At world w_{-3} A and B are both false and C true, whereas at w_2 A and B are both true and C false. The functions which build or modify *worldvals* are init-worldvals, update-wvals, wigamma, upd-worldvals.

Wrldvals has the same form as *worldvals*. In fact it is initialized to *worldvals*. Once an arrow set has been determined to form an RTFC configuration, the values which must hold in order for the added arrows to be consistent (from *necvals*) are added to *wrldvals*. If that arrow set is found to be inconsistent, then *wrldvals* is re-initialized to *worldvals* and the next arrow set generated and tested. If, on the other hand, the configuration is found to be consistent, *wrldvals*, *arwset*, and *path* are returned as a false model. The functions which build or modify *wrldvals* are upd-wrldvals, update-newvals.

eqtemp

Data structure *eqtemp* is the equivalence class template from which a configuration template (and subsequently an *arwset*) is generated. *Eqtemp* is initialized to one equivalence class containing all worlds in the configuration (excluding w_1). Once all possible configurations have been generated, *eqtemp* is incremented. For example, if there are three worlds (excluding w_1) in the current semi-complete structure, the successive values of *eqtemp* are:

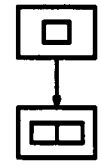
(1) (3)

(2) (1 2)

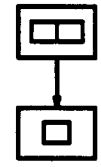
(3) (2 1)

(4) (1 1 1)

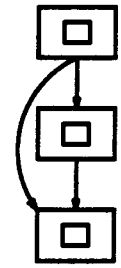
Eqtemp (1) consists of one equivalence class containing all three worlds. *Eqtemp* (2) consists of two equivalence classes, the first containing one world, the second two worlds:



Eqtemp (3) also consists of two equivalence classes, the first containing two worlds and the second one world:



Eqtemp (4) consists of three equivalence classes, one world in each:



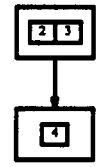
The functions which build or modify *eqtemp* are init-eqtemp, next-eqtemp.

config

Data structure *config* is the current configuration of worlds (generated from *eqtemp*) from which the set of added arrows is generated. So if the current *eqtemp* is (2 1) and worlds on the current path are w_2, w_3, and w_4, then *config* has the following successive values:

((2 3) (4))
((2 4) (3))
((3 4) (2))

The values of *config* represent all possible ways in which the world labels can be fitted into the current *eqtemp*. The first *config* above represents a configuration comprised of two equivalence classes, the first containing worlds w_2 and w_3, the second containing world w_4:



The functions which build or modify *config* are init-config, init-class, next-config, next-world.

arwset

Data structure *arwset* is the set of arrows added to the semi-complete structure to form an RTFC configuration. It is generated from *config* in the following way:

Algorithm Arrowset

1. *arrowset* ← {}
2. Repeat
   2.1 For each w_i in topmost EQ class do
      (a) For each w_j, $i \neq j$, in topmost and lower EQ classes do
         arrowset ← *arrowset* ∪ $\{Aw_iw_j\}$
   2.2 Remove topmost EQ class

Until no EQ classes left

Thus the set of arrows generated from *config* ((2 3) (4) (5 6)) is {(Aw_2w_3), (Aw_3w_2), (Aw_5w_6), (Aw_6w_5), (Aw_2w_4), (Aw_2w_5), (Aw_2w_6), (Aw_3w_4), (Aw_3w_5), (Aw_3w_6), (Aw_4w_5), (Aw_4w_6)}. The functions which build or modify *arwset* are get-arwset, update-arwset, c1, ci, upd-arwset.

alterns

Data structure *alterns* is a list of alternative value assignments to propositional variables at each world in the configuration. Variable assignments which must hold (from *wrldvals*) are appended to the alternatives for the remaining unassigned variables. This structure is designed to handle situations in which application of the rule for alternatives was delayed. In some cases arrow(s) force assignments to β_1, β_2, for some β-formula, but in others they do not. *Alterns* provides the alternative assignments without generating all the alternative SCS. Consider an example:

```
((1  ((B 1) (C 1) (A 0))
     ((B 1) (C 1) (A 1)))
 (2  ((A 1) (B 1) (C 0)))
 (-3 ((A 1) (C 0) (B 1))) ).
```

B and *C* must both be true at world w_1, but *A* may be true or false. There is only one alternative value assignment to variables at worlds w_2 and w_{-3} above. The functions which build or modify *alterns* are get-alternatives, get-alterns, get-alts, upd-alterns, add1base2.

4. Complexity

The present section is concerned with the analysis of the computational complexity of algorithm N-Theorem-Prover. The following variable definitions are used in the analysis. Let:

k = the number of symbols in *wff*

f = the degree of *wff*

b = the β-degree of *wff*

a = the α-degree of *wff*

g = the γ-degree of *wff*

n = the number of worlds in the structural template, omitting w_1

p = the number of worlds in SCS created due to a true ⇒-subformula

q = the number of EQ classes in *eqtemp*
C_i = the i^{th} EQ class in *eqtemp*
s_i = the cardinality of C_i

Note that:

(1) $f = a + b + g$

(2) $n \leq g$, $n = g$ in templates containing all p worlds created by the rule for a true $\Rightarrow$.

(3) $p \leq n$, $p = n = g$ when no worlds have been created by the rule for a false $\Rightarrow$.

The complexity of step 1 of the algorithm is $O(k)$ to put *wff* in prefix notation. Step 2 builds $O(3^b)$ SCS each containing $n + 1$ worlds. The modified rule for alternatives creates fewer SCS, but in the worst case each β-subformula consists of γ-subformula forcing the construction of $O(3^b)$ SCS. Thus the complexity of step 2 is $O(3^b n)$.

Step 3 generates and tests configurations. In the worst case all possible configurations must be generated. The outer loop is repeated for each of the $O(3^b)$ SCS. Step 3.1 generates structural templates from the current SCS. Each of the p worlds created by the rule for a true $\Rightarrow$ extends the generating tree by adding two branches to each leaf. Thus $O(2^p)$ structural templates are generated. However, $p = n$ in the worst case, so the complexity of step 3.1 is $O(2^n)$.

Step 3.2 tests each template for consistency. Aw_1w_j' is tested, $1 \leq j \leq n$, as well as Aw_iw_j' for nested arrows Aw_iw_j. Each test is $O((f' - b' + 3^{b'}a')n)$, and the test is performed on $O(2^n)$ templates.

Steps 3.3 and 3.4 are done in constant time. Step 3.5 builds *necvals* and *arrows*. Construction of *necvals* requires testing $\binom{n}{2}$ arrows where each test is $O(f' - b' + 3^{b'}a')$, whereas *arrows* requires comparison of r value assignments for n sets of arrows with $n - 1$ arrows in each set. Thus the complexity of step 3.5 is $O(n^2(f' - b' + 3^{b'}a') + nr(n - 1))$.

The repeat loop of step 3.6 is iterated for each of $O(2^n)$ structural templates, and step 3.7 for $q = 1$ to n classes in *eqtemp* ($O(n)$). Step 3.8 is repeated for each *eqtemp* of q classes. The number of possible *eqtemps* of q classes is an arithmetic progression from 1 to $n - q + 1$. Each s_i can be at most $n - q + 1$, where s_i is the cardinality of EQ class C_i. The sum of all s_i in an *eqtemp* must equal n. The sum of an arithmetic progression is given by:

$$S = \frac{[a_1 + a_n] \times n}{2}$$

where a_1 is the first term, a_n the last term, and n is the number of terms in the progression. Thus we obtain:

$$S = \frac{[1 + n - q + 1] \times (n - q + 1)}{2}$$

$$= \frac{(n - q + 2)(n - q + 1)}{2}$$

possible *eqtemps* of q classes.

Step 3.9 is repeated for each *config* generated from an *eqtemp*. There are $\binom{n}{s_1}\binom{n-s_1}{s_2} \cdots \binom{s_q}{s_q}$ possible ways of fitting n world labels into an *eqtemp* of q classes and s_i worlds within class C_i, $1 \le i \le q$. Generating and testing *arwset* is $O(\binom{n}{2})$. Thus the complexity of algorithm N-Theorem-Prover is given by:

$$\binom{n}{2}\left[\binom{n}{s_1}\binom{n-s_1}{s_2} \cdots \binom{s_q}{s_q}\right]\left[\frac{(n - q + 2)(n - q + 1)}{2}\right] n\, 2^n\, 3^b$$

$$= O\left(2^n\, 3^b\, n^3\left[\binom{n}{s_1}\binom{n-s_1}{s_2} \cdots \binom{s_q}{s_q}\right]\left[\frac{(n - q + 2)(n - q + 1)}{2}\right]\right)$$

Note that as q approaches 1 the fourth term approaches $O(n^2)$, and the third term approaches n (when $q = 1$, the third term is 1). As q approaches n the third term approaches $n!$, and the fourth term approaches 1. Thus algorithm N-Theorem-Prover is exponential.

Chapter 5. Conclusions

1. Summary

This thesis has presented a tableau-based approach to theorem proving for a conditional logic. Logic N extends classical propositional logic by adding a variably strict conditional operator $\Rightarrow$. The formal semantics for N is based on a possible worlds approach. The basis of the accessibility relation among possible worlds is uniformity, or "unexceptionalness," i.e., accessibility relation E is said to hold between two worlds w_1 and w_2 just when w_2 is at least as unexceptional as w_1.

The theorem prover is a refutation procedure based on the model theory of N rather than the axiomatic basis of N. An attempt is made to construct a falsifying model for a wff ω. If such a model can be constructed, then $\neg\omega$ is satisfiable and ω is invalid. The falsifying model serves as a counterexample. Otherwise a falsifying model is shown to be impossible. In this event $\neg\omega$ is unsatisfiable, and ω is therefore valid. The *method of N-diagrams* is presented to accomplish model construction. The method utilizes rectangles representing worlds, arrows representing accessibility between worlds, and rules for building models based on the definition of truth in the model theory.

The approach is based on techniques of tableau and semantic diagrams. The method of semantic diagrams for modal logics such as *T*, *S*4, and *S*5 differs from the method of N-diagrams in a significant way, though, in part because of the different accessibility relations in these systems. The accessibility relation for *T* is reflexive, for *S*4 reflexive and transitive (but not forward-connected), and for *S*5 reflexive, transitive, and symmetric (fully connected). As a result, there is a unique system of diagrams for these logics and the diagrams are complete after applying the rules of the method as often as possible. However, the accessibility relation E for N is reflexive, transitive, and forward-connected (between *S*4 and *S*5). Once the construction rules have been applied as often as possible, the system of N-diagrams is only semi-complete. Forward-connectedness and transitivity must be imposed. The semi-complete structure is then used to generate RTFC configurations by adding RTFC sets of arrows to templates of the semi-complete structure.

What this means is that diagrams in a system of more than one semantic diagram differ only in alternative value assignments to the operands of the β-subformula within rectangles. New diagrams are only required when the rule for alternatives is applied. New diagrams are similarly required in a system of N-diagrams. Additionally, though, the conditions for a true γ-subformula in N contain an alternative. It is from these alternatives that we obtain the templates from a semi-complete structure. Further, the RTFC patterns of arrows must be added to the templates. Thus diagrams in a system of N-diagrams may contain different labelled rectangles and different patterns of arrows connecting labelled rectangles.

Both methods employ tableau techniques. Applying the modified rule for alternatives, we in effect obtain a tableau within each rectangle. However, the unmodified rule for alternatives creates different diagrams, one for each alternative value assignment to the operands of the β-subformula, whereas a tableau branches to the left and to the right in the case of a β-subformula.

The approach provides a simple, pictorial procedure for proving wffs of N. The approach has been implemented in program *Validate*, an automated theorem prover for logic N. Full documentation is presented in Appendix 2. Part I provides a manual for users of the program, part II a manual for maintaining the program. The computational complexity of the algorithm has been shown to be exponential. This is as expected because all known algorithms for propositional logic are exponential, and N subsumes propositional logic.

Logic N provides an approach to the representation of knowledge about prototypes and prototypical properties of kinds. The exception-admitting quality of such knowledge poses some special representation problems. Not every member of the kind possesses all the properties associated with the kind. It seems that, in general, models of how things work are more easily formulated than models of how things "go wrong." It has been argued that logic N deals with these problems more appropriately than other approaches in AI. The logic is, however, without modus ponens. Delgrande [87] has presented an approach in which modus ponens can be effectively used.

2. Further Research

Some areas for further work are: (1) extension of the theorem-proving approach to first-order N, (2) application of the approach to other conditional logics, and (3) improvements to the arrow-set generation process. The extension to first-order requires implementation of the first-order rules given in Appendix 1. Quantification applies within one world only, so the extension is straightforward. A first-order augmentation would further require some kind of heuristics or restrictions to handle the loss of decidability. The heuristics used by Oppacher and Suen [86] in their implementation of a tableau-based first-order theorem prover could be used. The goal with a first-order knowledge base is often to reason about individuals, and in this case we have a decision procedure for individuals.

The implementation is easily extended to other normal conditional logics such as the logic for counterfactual conditionals of Lewis. It is only the last component dealing with the accessibility relation E which applies directly to N. The rule for new worlds remains unchanged because the truth conditions for the variable conditional are the same for the class of normal conditional logics.

An enhancement of the arrow-set generation process might be feasible by reconfiguring the generator. In applying the method of N-diagrams it was demonstrated that in the case of valid wffs, validity can be derived from the table of arrow constraints. The automated theorem prover does not currently make use of this characteristic. No such derivation is, of course, possible for invalid wffs. Thus an implementation would require some heuristics for deciding when to give up trying.

The goal is to use the information in the table of arrow constraints to build configurations which might be consistent rather than enumerating all possible configurations in some order. For example, if "Aw_iw_j cannot exist" is in the table, then no consistent configuration contains arrow Aw_iw_j, and every consistent configuration contains Aw_jw_i. The goal is to modify the generator so that the only sets of arrows generated contain Aw_jw_i and omit Aw_iw_j.

A possible approach is to replace the single enumerative generator with a hierarchy of generators so that reconfiguration of the generation process on the basis of *a priori* information is possible. The table of arrow constraints provides us with *a priori* information regarding arrow subsets which must coexist, cannot coexist, or single arrows which must exist or cannot exist. The goal is to reconfigure the generating system

so that those arrow sets characterized by the exclusion information are not generated. The well-ordering on equivalence classes of worlds with respect to the accessibility relation makes such an approach attractive. Alternatively, or in addition, arrow sets can be characterized by inclusion information. A similar approach has been proposed for enumerative learning systems in [Holte and Wharton 86].

Another area that warrants further investigation is the separation of representation and reasoning mechanisms. Perhaps some conditional logics are more suited to the representation of exception-allowing information whereas others are more appropriate for reasoning about individuals. If translation between logics is straightforward, as in the case of N and default logic, this might be a viable approach for common-sense reasoning systems. The following discussion provides some background on conditional logics in general.

The early view of conditional statements of the form "if A then B" was one of class inclusion. The following historical perspective is from [Jennings 87]. According to this perception, "if A then B" is true just when $\{A\} \subseteq \{B\}$, where $\{P\}$ is the set (or "proposition") representing P, the set of "occasions," "worlds," "situations," or "models," in a universe of possible states ("occasions," "worlds," etc.) in which P is true. This semantics is illustrated with the Venn diagrams of Figure 5.1. A first-order representation for inclusion is given by:

$$\forall x, x \in \{A\}\ \exists y, y \in \{\mathrm{B}\} \wedge x = y.$$

Material conditionals exhibit the following properties:

(a) transitivity: $((A \supset B) \wedge (B \supset C)) \supset (A \supset C)$

(b) modus ponens: $A, A \supset B \vdash B$

(c) contraposition: $(A \supset B) \equiv (\neg B \supset \neg A)$

(d) law of the excluded middle: $(A \supset B) \vee (A \supset \neg B)$

(e) confirmation:

(1) $(A \supset B) \supset ((A \vee C) \supset (B \vee C))$

(2) strengthening the antecedent: $(A \supset C) \vdash (A \wedge B) \supset C$

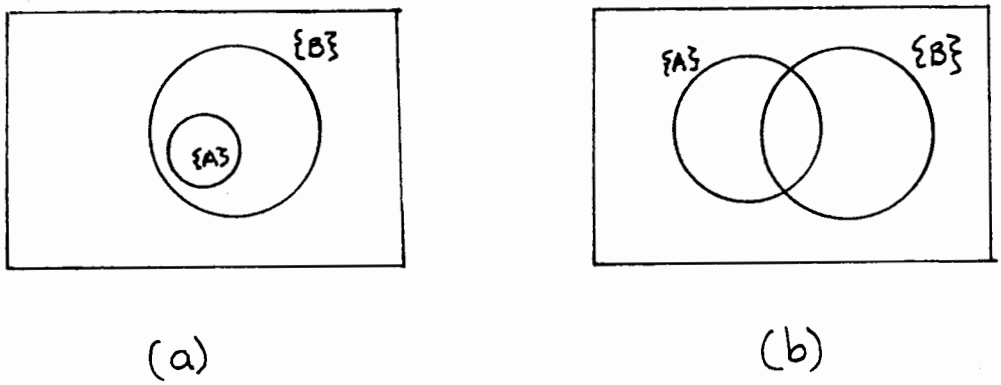


Figure 5.1 Inclusion representation of the material conditional. $A \supset B$ is true in (a) and false in (b).

(3) weakening the consequent: $(A \supset B) \vdash A \supset (B \vee C)$

(4) $(A \supset (B \wedge C)) \supset ((A \wedge B) \supset C)$

(f) strong deduction: $((A \wedge B) \supset C) \equiv (A \supset (B \supset C))$

The "is included in" relation is clearly transitive as demonstrated in Figure 5.2(a). Figure 5.2(b) illustrates confirmation property (2), strengthening the antecedent. This property of the material conditional allows us to conclude that $(A \wedge B) \supset C)$ is true whenever $A \supset C$ is true. In terms of Venn diagram (b):

(1) If $A \supset C$ then $\{A\} \subseteq \{C\}$.

(2) $\{A\} \cap \{B\} \subseteq \{A\}$.

(3) Thus by transitivity $\{A \wedge B\} \subseteq \{C\}$.

Figure 5.2(c) illustrates confirmation property (3), weakening the consequent. This property of the material conditional allows us to conclude $A \supset (B \vee C)$ whenever $A \supset B$ is true. In terms of Venn diagram (c):

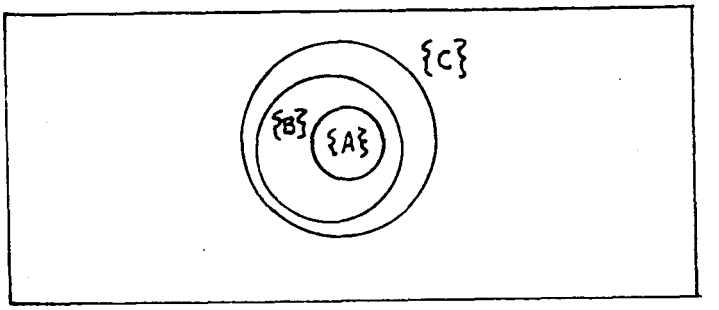
(1) If $A \supset B$ then $\{A\} \subseteq \{B\}$.

(2) $\{B\} \subseteq \{B\} \cup \{C\}$.

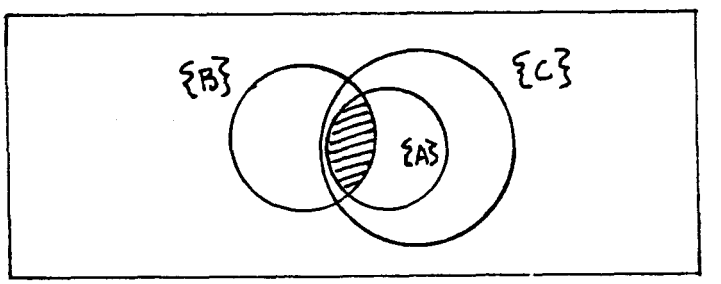
(3) Thus by transitivity $\{A\} \subseteq \{B \vee C\}$.

As a representation, the material conditional is fine for (1) terms which have analytic definitions such as "bachelor" or "square," (2) nouns derived by transformation from verbal forms such as "hunter = one who hunts," and (3) statements such as "penguins are birds." In cases (1) and (2) $\{A\} \subseteq \{C\}$ and $\{C\} \subseteq \{A\}$. For example, let $\{A\}$ represnt the set of squares and $\{C\}$ the intersection of the set of plane figures, the set of objects with four equal sides, and the set of objects with four right angles. For statements of type (3) $\{A\} \subseteq \{C\}$ and $\{C\} \subseteq \{A\}$, e.g., $\{$*creature–with–heart*$\}$, $\{$*creature–with–kidneys*$\}$, or $\{A\} \subset \{C\}$, e.g., $\{$*penguin*$\} \subset \{$*bird*$\}$.

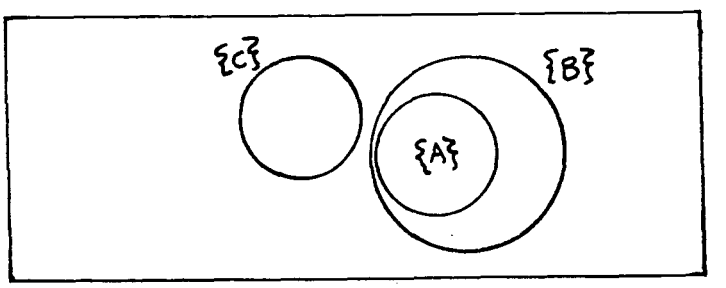
However, some if-then statements are arguably not intended to be understood as universal law. For example, "ravens are black," "water boils at 100° C," rules of thumb, or "natural language conditionals" [Jennings 83] such as "If I strike this match it will light." In these cases we want to allow exceptions without falsifying the conditional. Let $\{A\}$ be the set representing ravens, $\{B\}$ the set representing things



(a)



(b)



(c)

Figure 5.2 Properties of the material conditional: (a) transitivity, (b) strengthening the antecedent, and (c) weakening the consequent.

that are black, and $\{C\}$ the set representing albino things. It is not the case that $\{A\} \subseteq \{B\}$ as is illustrated in Figure 5.3. For example, albino ravens are represented in $\{A\}$ but outside $\{B\}$, (though it is not necessarily the case that $\{C\} \cap \{B\} = \emptyset$, e.g., an albino raven which becomes the victim of an oil spill). Statements of this kind exhibit the property of strengthening the antecedent and obtaining the negation of the consequent, referred to as *left-downward nonmonotonicity*. This property is illustrated in Figure 5.4, where $\{A\}$, $\{B\}$, $\{C\}$ are as above.

Recall the inclusion representation for $A \supset B$:

$$\forall x, x \in \{A\}\ \exists y, y \in \{B\} \wedge x = y.$$

The problem is the universal quantification of elements of A. Three approaches to this problem may be taken:

(1) restrict the antecedent set $\{A\}$,

(2) extend the consequent set $\{B\}$,

(3) weaken identity.

Approach (1) restricts the antecedent set $\{A\}$ to members of $\{A\}$ which are included in $\{B\}$. Thus subsets of $\{A\}$ are considered where B imposes the restrictions on A, i.e., "if a preferred set of A then B." Approach (2) extends the consequent B to include all members of $\{A\}$. Thus a superset of B is considered where A determines how B is extended.

DeMorgan in 1864 proposed a semantic approach which if applied to the inclusion representation yields (3):

$$\exists R \in G\colon\ \forall x, x \in \{A\}\ \exists y, y \in \{B\} \wedge xRy,$$

where G is a set of relations one of which may be identity, and R is a relation of G. Another possibility is to drop the restriction that the same R must relate elements of $\{A\}$ and $\{B\}$.

Each approach yields a different kind of logic. Conditional logics based on semantical approach (1) are referred to as *ldn logics* (left-downward nonmonotonic), and those of (2) *run logics* (right-upward nonmonotonic). Approach (3) yields quantum logic. (For more on the conditional in quantum logic see [Hardegree 76] [Mittelstaedt 78].)

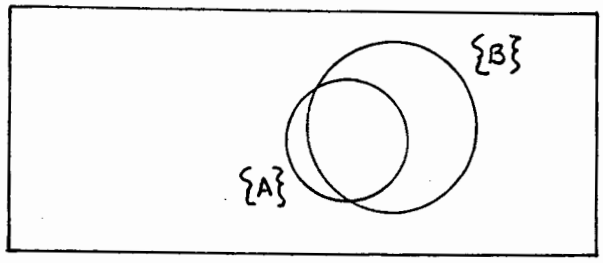


Figure 5.3 An example in which $A \supset B$ is false.

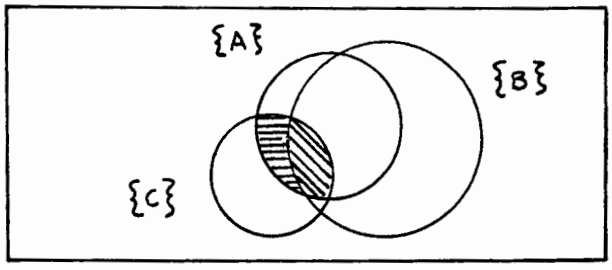


Figure 5.4 Left-downward nonmonotonicity.

The choice of an approach depends upon the particular application. Perhaps a ldn logic is more appropriate for the *representation* of properties of kinds, relations between kinds, and natural laws, whereas a run logic might be more suited to diagnostic or default *reasoning*. For an example of the former, consider the statement "water boils at 100° C." Let $\{A\}$ be the set representing water samples and $\{B\}$ the set representing things that boil at 100° C. There may be particular samples of water which boil at lower temperatures, those with a high mineral content, for example. It seems more natural to restrict the set $\{A\}$ to pure samples than to extend $\{B\}$ to include things that do not boil at 100° C. Perhaps there are associated with such statements implicit assumptions that the water sample is pure, the pressure at a certain level, etc. If such assumptions are made then we are in effect restricting the antecedent. Similarly, we might prefer to restrict the set representing ravens rather than extend the set representing black things to include white things in a treatment of "ravens are black."

Alternatively, a run logic may be more suited to reasoning in diagnostic or default systems. For example, a medical system returns a diagnosis given a set of symptoms. It seems more natural to say something like "this set of symptoms indicates disease $B \vee C \vee D$" as opposed to restricting the set of symptoms. Similarly, a default system may be asked to predict properties of an individual given its kind. A semantical approach which allows disjunction of predictions may be preferred.

Appendix 1. Tableau Branch-Extension Rules for Modal and First-order Logics

An extension to the tableau method to handle the modal operators □ (necessity) and <> (possibility) due to [Fitting 83] are shown in Table A2.

Table A1. Modal rules for the assignment of truth values.

ν	ν_o	π	π_0
$T\Box X$	TX	$F\Box X$	FX
$F\Diamond X$	FX	$T\Diamond X$	TX

Note that $\models^M_w \nu \equiv Ewv_1, \forall w_1 \models^M_{w_1} \nu_0$ and $\models^M_w \pi \equiv Ewv_1, \exists w_1 \models^M_{w_1} \pi_0$.

The extension to first-order logic requires two rules in addition to the α- and β-rules [Fitting 83]:

Table A2. Rules for first-order quantifiers.

γ	$\gamma(a)$	δ	$\delta(a)$
$T(\forall x)A(x)$	$TA(a)$	$T(\exists x)A(x)$	$TA(a)$
$F(\exists x)A(x)$	$FA(a)$	$F(\forall xA(x)$	$FA(a)$

Fitting's γ-rules apply for any constant a, and the δ-rules for any parameter new to the branch.

Appendix 2. Program VALIDATE Manual

The purpose of program VALIDATE is to determine the validity of well-formed formulas (wffs) expressed in conditional logical system N [Delgrande 86]. Since N subsumes propositional logic, the program handles wffs of propositional logic as well. A wff of N is given as input, and VALIDATE returns either a message stating that the given wff is valid or a message stating that the given wff is invalid. If invalid, a set of value assignments to variables is also returned in which the wff evaluates to false.

The algorithm is an implementation of the method of N-diagrams, a tableau-based approach with modifications to handle the possible worlds semantics of logic N. An attempt is made to construct a falsifying model. If such a model can be constructed, the given wff is invalid. Otherwise such a model is shown to be impossible to construct, and the wff is therefore valid. The program is written in *Franz Lisp* code and is intended to be run on the *Franz Lisp* interpreter.

The manual is divided into two main parts: (1) using the program, and (2) maintaining the program. Part I concerns the use of program VALIDATE. It is intended to be comprehensible to readers with no knowledge of computer programming. Section 1 provides an overview of logic N and the method of N-diagrams. Instructions for entering data are given in section 2, for interpreting the output in section 3, and for running the program in section 4. Section 5 provides sample input/output.

Part II provides information helpful for program maintenance. It is assumed that the reader of part II has programming knowledge, of *Lisp* in particular. Section 1 of part II consists of a brief outline of what the program does. The algorithm is presented in section 2. Section 3 presents the functions arranged in categories according to the type of work performed. There are three main types of functions in program VALIDATE: (1) those which build or modify data structures, (2) those which perform tests, and (3) those which manipulate lists. The last section of part II provides a description of each function.

PART I. USING THE PROGRAM

1. Introduction

The purpose of program VALIDATE is to determine the validity of well-formed formulas (wffs, see Section 2) expressed in conditional logical system N [Delgrande 86]. Since N subsumes propositional logic, the program handles wffs of propositional logic as well. A wff of N is given as input, and VALIDATE returns either a message stating that the given wff is valid or a message stating that the given wff is invalid. If invalid, a set of value assignments to variables is also returned in which the wff evaluates to false. The program is written in *Franz Lisp* code and is intended to be run on the *Franz Lisp* interpreter.

1.1. Logic N

System N provides a propositional logic with the addition of a variably strict conditional operator $\Rightarrow$ where $A \Rightarrow B$ is interpreted as "all other things being equal, if A then B," or "in the normal course of events, if A then B." Consider a statement such as "Ravens are black." Representing this statement with the material conditional operator $\supset$ of propositional logic, we obtain $Raven \supset Black$ which requires that every raven be black. This does not precisely capture the intension of the statement. $Raven \Rightarrow Black$, interpreted as "in the normal course of events, if a thing is a raven then it is black," on the other hand, allows for albino ravens, featherless ravens, etc. The $\Rightarrow$ operator seems more suited to the representation of knowledge of this kind.

The semantics for the variable conditional operator $\Rightarrow$ rests upon the notion of possible worlds and an accessibility relation between worlds. More formally, the accessibility relation E is said to hold between two worlds w_1 and w_2 (Ew_1w_2) just when w_2 is at least as unexceptional as w_1. $A \Rightarrow B$ is true with respect to a world w_1 just when (1) the least exceptional worlds which have A true have B true also, and in no less exceptional world is $A \supset B$ false, or (2) A is false at all worlds accessible from w_1. $A \Rightarrow B$ is false at a world w_1 just in case there exists some world w_2 accessible from world w_1 where A is true and B false and either (1) there exists no world w_3 accessible from w_2 where A is true and B is true, or (2) if there is an

accessible world w_3 where A and B are both true, then w_2 is also accessible from w_3. The accessibility relation is reflexive, transitive, and forward-connected. The property of forward-connectedness states that if Ew_1w_2 and Ew_1w_3 then either (1) Ew_2w_3, (2) Ew_3w_2, or (3) both Ew_2w_3 and Ew_3w_2 hold, i.e., any two worlds accessible from the same world are themselves related.

Other characteristics of the logic are:

(1) strengthening of the antecedent and a possible associated change in the consequent to its negation,

(2) failure of transitivity,

(3) a weakened form of transitivity, and

(4) lack of a standard law of the excluded middle.

For example,

(1) $\{A \Rightarrow B, A \wedge C \Rightarrow \neg B\}$ is satisfiable,

(2) $\{A \Rightarrow B, B \Rightarrow C, \neg (A \Rightarrow C)\}$, $\{A \supset B, B \Rightarrow C, \neg (A \Rightarrow C)\}$ are satisfiable,

(3) "If $(A \Rightarrow B)$ and $\models (B \supset C)$ then $(A \Rightarrow C)$" is valid, and

(4) $\neg ((\neg (A \Rightarrow B)) \supset (A \Rightarrow \neg B))$ is satifiable while $\neg \Box \neg A \supset ((A \Rightarrow B) \supset \neg (A \Rightarrow \neg B))$ is valid.

1.2. Method of N-diagrams

Program VALIDATE is a tableau-based theorem-prover for conditional logic N. An attempt is made to construct a falsifying model for a given wff. If such a model can be constructed, then the wff is invalid. Otherwise it is shown that such a model is impossible to construct, and the wff is therefore valid.

The method of N-diagrams is used in the construction of models. A system of N-diagrams consists of:

(1) rectangles,

(2) labels,

(3) arrows,

(4) not-arrows,

(5) ORs, and

(6) rules for building the diagrams.

A rectangle represents the relevant state of affairs or conditions which hold at some world. Figure 1 represents the truth conditions for the variable conditional operator ⇒ using N-diagrams.

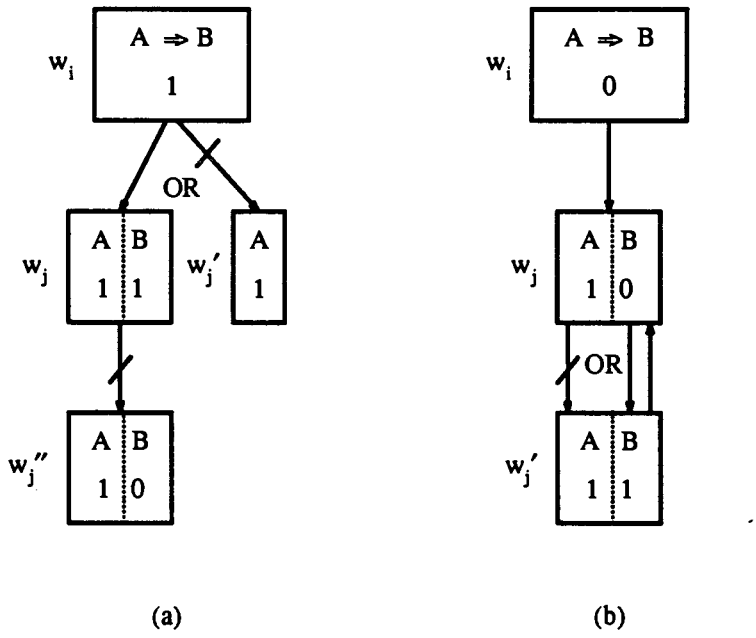


Figure 1. N-diagrams representing the truth conditions for the ⇒ operator. Figure 1(a) illustrates the conditions under which $A \Rightarrow B$ is true at a world w_i. Figure 1(b) shows the conditions under which $A \Rightarrow B$ is false at w_i.

The rectangle labelled w_i in Figure 1(a) represents a world w_i in which $A \Rightarrow B$ is true, while Figure 1(b), represents a world w_i in which $A \Rightarrow B$ is false. An arrow from one world (rectangle) to another indicates that the world at the head of the arrow is accessible from the world at the tail of the arrow. So world w_j is accessible from w_i in Figure 1(b), i.e., Ew_iw_j holds in Figure 1(b). There is an implicit arrow from each rectangle to itself because the accessibility relation is reflexive. A not-arrow, $\not\rightarrow$, indicates that there can be no world accessible from the world at the tail of the not-arrow in which the conditions hold represented in the rectangle at the head of the not-arrow. For example, in Figure 1(a) there can be no world accessible

from world w_j in which A is true and B false. The OR between an arrow and a not-arrow or between a not-arrow and a pair of double-arrows (an arrow and its return-arrow) indicates that either one accessibility path or the other must hold. Thus in Figure 1(a) either (1) there is a world w_j accessible from w_i where A and B are both true, and no world accessible from w_j where A is true and B false, OR (2) there is no world accessible from w_i in which A is true. In Figure 1(b) either there is no world w_j' accessible from w_j where A and B are both true OR if there is such a world w_j' then w_j must also be accessible from w_j', i.e., there must also be an arrow back to w_j from w_j'.

There are two rules for creating new worlds according to the truth conditions for the $\Rightarrow$ operator:

(1) If an asterisk occurs beneath a $\Rightarrow$ assigned 1 at w_i, create a new rectangle w_j in which the antecedent is true and the consequent is true. (Refer to Figure 1(a).) Place an arrow from rectangle w_i to new rectangle w_j. Create another rectangle w_j' in which the antecedent is true. Place a not-arrow from rectangle w_i to rectangle w_j'. Place an OR between the arrow from w_i to w_j and the not-arrow from w_i to w_j'. Create another rectangle w_j'' in which the antecedent is true and the consequent false. Place a not-arrow from rectangle w_j to rectangle w_j''.

(2) If an asterisk occurs beneath a $\Rightarrow$ assigned 0 at w_i, create a new rectangle w_j in which the antecedent is true and the consequent false. (Refer to Figure 1(b).) Place an arrow from rectangle w_i to rectangle w_j. Create another rectangle w_j' in which the antecedent and the consequent are both true. Place a not-arrow from rectangle w_j to rectangle w_j'. Place an arrow from rectangle w_j to rectangle w_j' and from rectangle w_j' to rectangle w_j. Place an OR between the not-arrow and the double-arrows connecting w_j and w_j'.

The algorithm for deciding N-validity is given in Algorithm N-DIAGRAMS.

The procedure is demonstrated with two examples. Consider the wff $(((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow (B \wedge C)))$, where $\wedge$ is the logical and operator, $\supset$ the material conditional operator, and $\Rightarrow$ the variable conditional operator. Executing step one we get:

w_1 | $(((A \Rightarrow B) \wedge (A \Rightarrow C)) \supset (A \Rightarrow (B \wedge C)))$ | 0

Algorithm N-DIAGRAMS

Input: wff

Output: valid, invalid

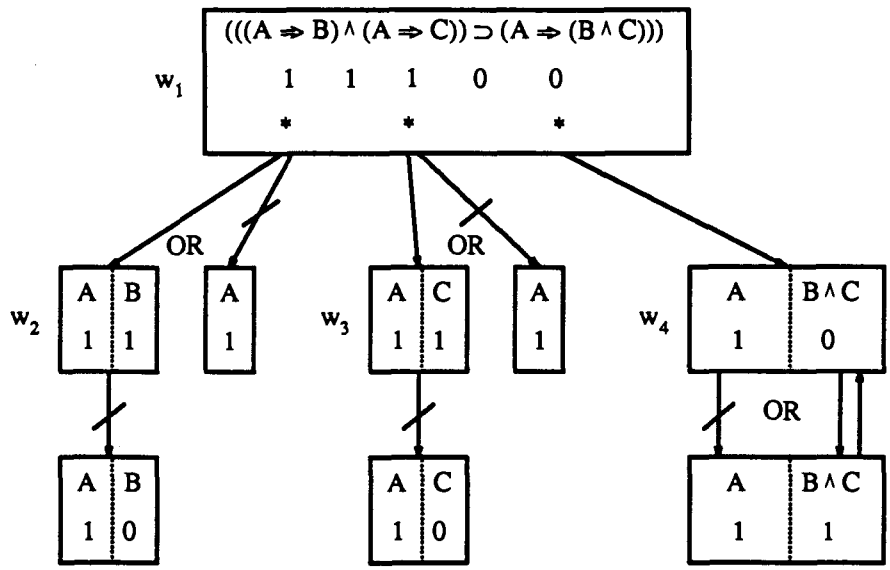
1. Initialize the system of N-diagrams
    1.1. Write the wff in a rectangle
    1.2. Label the rectangle w_1
    1.3. Assign 0 to the main operator of the wff
2. Build the semi-complete structure(s)
    2.1. Repeat
        Apply α-rules
        Apply rules for crosses, asterisks, and modified rule for alternatives
        Apply rule for new worlds
    Until rules applied as often as possible
3. Repeat {for each semi-complete structure}
    Generate an RTFC configuration
    Test the configuration
  Until all RTFC configurations tested or a consistent configuration found
4. If all configurations have been tested and found inconsistent
    Then return VALID
    Else return the consistent configuration

Step two involves the application of propositional rules for assignment of truth values to subformulas. Whenever a ⇒ operator is assigned a value, an asterisk is placed beneath the assigned value:

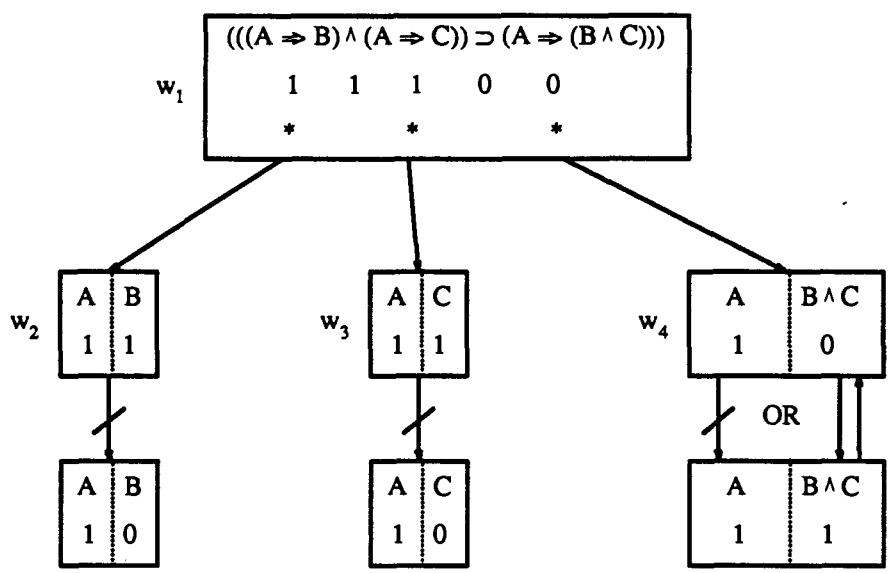
w_1

(((A ⇒ B) ∧ (A ⇒ C)) ⊃ (A ⇒ (B ∧ C)))					
	1	1	1	0	0
	*		*		*

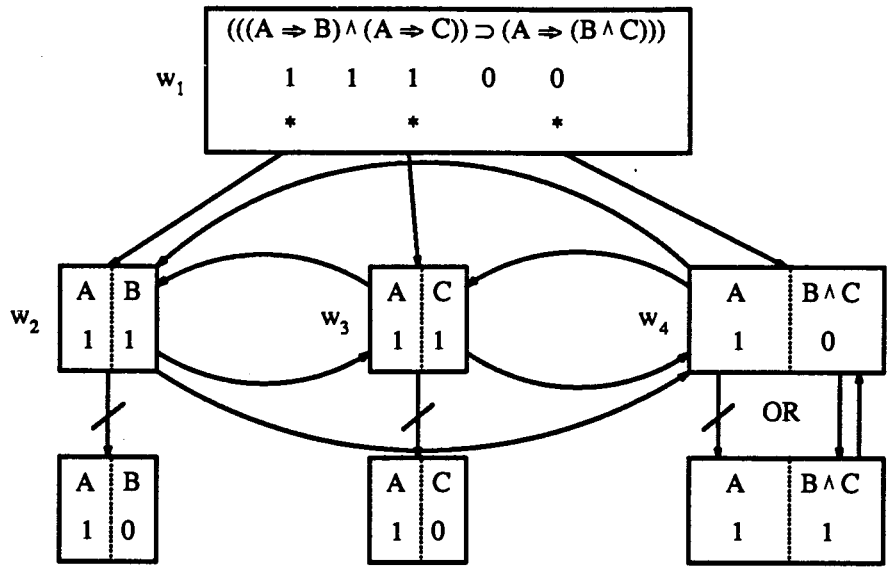
One is assigned to the antecedent of the false implication above and 0 to the consequent. One is assigned to both operands of the true logical and in the antecedent. Applying the rules for creating new worlds we obtain:



Observe that the not-arrows alternative to worlds w_2 and w_3 are inconsistent with the arrow to world w_4. Thus worlds w_2, w_3, w_4 must occur in any consistent N-diagram, or configuration:



Recall that the accessibility relation is reflexive, transitive, and forward-connected (RTFC). Therefore arrows must be added to the structure above in order that the RTFC properties hold. Step four generates all possible sets of arrows to create RTFC configurations and tests them. The fully-connected configuration is generated first:

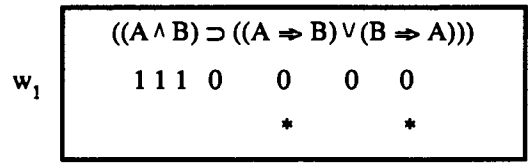


The above configuration is inconsistent since the arrow from w_2 to w_4 (Aw_2w_4) requires that B have value 1 and C 0 at w_4 whereas arrow Aw_3w_4 requires that C have value 1 and B 0 at w_4. Thus Aw_2w_4 and Aw_3w_4 cannot co-occur in any consistent configuration. This observation eliminates RTFC configurations containing the following sets of arrows: $\{Aw_2w_4, Aw_2w_3, Aw_3w_4, Aw_4w_3\}$, $\{Aw_3w_2, Aw_3w_4, Aw_2w_4, Aw_4w_2\}$, $\{Aw_2w_4, Aw_3w_4, Aw_2w_3, Aw_3w_2\}$, $\{Aw_2w_3, Aw_2w_4, Aw_3w_4\}$, $\{Aw_3w_2, Aw_3w_4, Aw_2w_3\}$,$\{Aw_2w_3, Aw_2w_4, Aw_3w_2, Aw_3w_4, Aw_4w_2, Aw_4w_3\}$.

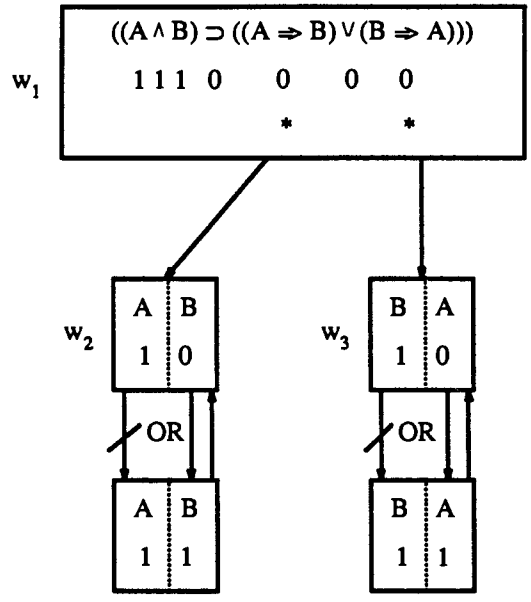
Observe also that if arrows Aw_3w_2 and Aw_4w_2 co-occur within a configuration, Aw_2w_4 must also appear in the configuration. This is so because Aw_3w_2 forces C to have value 1 at w_2, and Aw_4w_2 requires the return-arrow Aw_2w_4 since C is assigned 1 at w_2. This observation eliminates configurations containing the following sets of arrows: $\{Aw_4w_2, Aw_4w_3, Aw_2w_3, Aw_3w_2\}$, $\{Aw_3w_4, Aw_4w_3, Aw_3w_2, Aw_4w_2\}$, $\{Aw_3w_4, Aw_3w_2, Aw_4w_2\}$, $\{Aw_4w_3, Aw_4w_2, Aw_3w_2\}$. Similarly if arrows Aw_2w_3 and Aw_4w_3 co-occur, then Aw_3w_4 must also appear in the configuration. This eliminates the remaining sets of arrows which form RTFC configurations: $\{Aw_2w_4, Aw_2w_3, Aw_4w_3\}$, $\{Aw_4w_2, Aw_4w_3, Aw_2w_3\}$, $\{Aw_2w_4, Aw_4w_2, Aw_2w_3, Aw_4w_3\}$. No consistent RTFC configuration is possible, and the wff is valid.

Next consider the wff $((A \wedge B) \supset ((A \Rightarrow B) \vee (B \Rightarrow A)))$, where $\wedge$ is the logical and operator, $\supset$ the material conditional operator, $\Rightarrow$ the variable conditional operator, and $\vee$ the logical or operator.

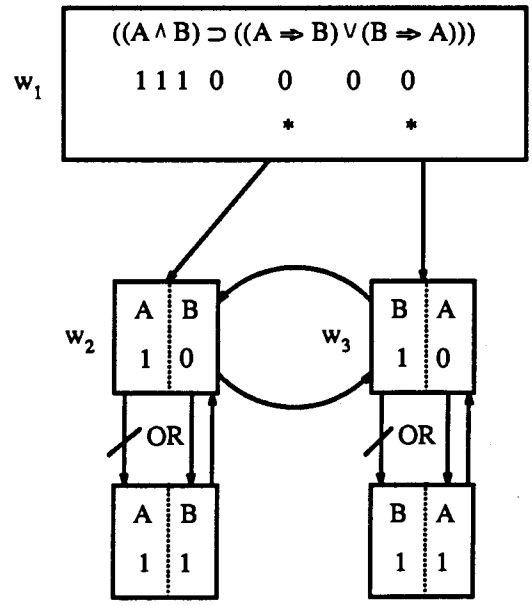
Execution of steps one and two gives:



Applying the rules for creating new worlds (step three), we obtain:



Next the fully-connected configuration is generated and tested. The set of arrows $\{Aw_2w_3, Aw_3w_2\}$ is added to the structure above to obtain:



Each arrow is then tested. Aw_2w_3 is consistent because A has value 0 at w_3, and Aw_3w_2 is consistent because B has value 0 at w_2. The configuration is RTFC, its arrows consistent, and its rectangles consistent. Thus a false model exists, and the wff is invalid. Program VALIDATE is an implementation of this method of N-diagrams for deciding N-validity.

2. Input

Program VALIDATE is a procedure for determining whether a given formula of logic N is valid. Thus the input to the program is a well-formed formula (wff) of N. The notion of wff is described by the following recursive rules:

(1) Any propositional variable (atomic formula) is a wff.

(2) If *A* is a wff, then so is (~ A).

(3) If A, B are wffs then so are (A ^ B), (A v B), (A @ B), (A => B), (A <=> B).

Six operations are known to the program. The symbols to be entered for the operators are in bold:

(1) **~** : negation

(2) **^** : conjunction, logical and

(3) **v** : disjunction, logical or

(4) **@** : material conditional

(5) **=>** : variable conditional

(6) **<=>** : equivalence.

A variable may be any *Lisp* atom other than the operators. There must be a space separating operators and variables, e.g., (~ A), not (~A). Wffs are entered in infix notation, i.e., *operand-operator-operand*, or *operator-operand*, if the operation is negation.

Wffs must be unambiguous. Parentheses are used to disambiguate wffs by placing a pair of parentheses around each occurrence of an operator and its operand(s), for example:

(~ ((A => C) ^ ((A ^ B) => (~ C))))

(~ A)

(A v (~ A))

((A => B) @ (((A ^ B) => C) @ (A => C))).

3. Output

The program returns one of two messages:

(1) The formula α is valid.

(2) The formula α is invalid.

In the case that a wff is determined to be invalid, a false model is also returned. There may be many different models in which the wff evaluates to false. The program returns the first one it finds. If even one false model exists, then the wff is invalid. For example, consider the wff ((((*A* => *B*) *v* (*C* => *B*)) ^(*A* ^*C*)) @ *B*). The program returns the following message:

```
The formula ((((A => B) v (C => B)) ^ (A ^ C)) @ B) is invalid.
A false model exists:
 world value-assignment
((1 (A 1) (C 1) (B 0) ((A => B) 1) ((C => B) 0))
 (-3 (C 1) (B 0) (A 0))
 (2 (A 1) (B 1) (C 0))))
 arrows
((A -3 2) (A 2 -3) (A 1 -3) (A 1 2))))
```

The input wff is echoed in the first line. The third line consists of column labels. So the numbers comprising the first column are world labels, whereas the second column consists of relevant value assignments to variables and/or ⇒–subformula.* The list following the heading "arrows" comprises the set of arrows indicating accessibility relations among the worlds. In the model above there is a world w_1 where *A* is true (has truth value 1), *B* is false, *C* is true, $A \Rightarrow B$ is true, and $C \Rightarrow B$ is false, a world w_{-3} where *C* is true, and *A* and *B* false, and a world w_2 where *A* and *B* are true and *C* false. World w_2 is accessible from w_1 and w_{-3},

* A ⇒–subformula is a subformula in which the main operator is the ⇒ operator.

whereas world w_{-3} is accessible from w_1 and w_2.

This information can be translated into a system of N-diagrams in the following way:

(1) Draw a rectangle for each world label in the world column.

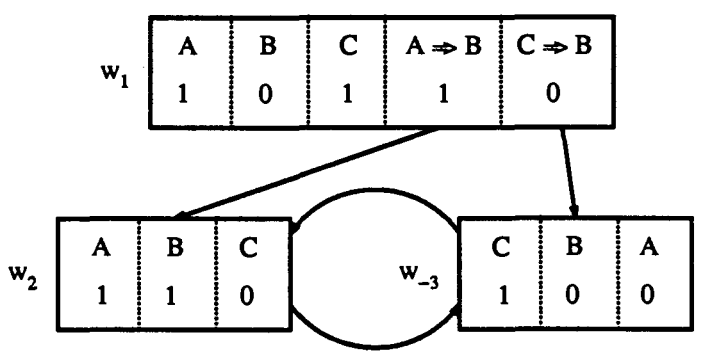
(2) Label the rectangles with the appropriate world labels.

(3) Place the value-assignments in the appropriate rectangles.

(4) For each arrow Aw_iw_j in the list of arrows, draw an arrow from rectangle w_i to rectangle w_j.

(5) For each not-arrow (w_i ($\sim\alpha$)) in the list of arrows, draw a rectangle. Place the subformula ($\sim\alpha$) in the rectangle. Place a 1 under the main operator. Draw a not-arrow from rectangle w_i to the newly created rectangle.

The system of N-diagrams for the example above is:



4. Running the Program

Program VALIDATE is run on a *Franz Lisp* interpreter. It may be run interactively or from a file of *Lisp* commands. A list of 42 example wffs have been entered into a file called *examples*. These formulas can then be referred to by their names f1-f42.

The following commands instigate an interactive session:

```
sfu_cmpt% lisp
Franz Lisp, Opus 38.91
-> (load 'cmufncs)
[fasl /usr/local/lib/lisp/cmufncs.o]
t
-> (load 'cmuenv)
```

```
[load /usr/local/lib/lisp/cmuenv.l]
[fasl /usr/local/lib/lisp/cmumacs.o]
[fasl /usr/local/lib/lisp/cmufncs.o]
[fasl /usr/local/lib/lisp/cmutpl.o]
[fasl /usr/local/lib/lisp/cmufile.o]
t
1.(dskin validate)
[load validate]
(validate)
2.(dskin examples)
[load examples]
(examples)
```

The first command causes a change from the operating system environment (Unix in this case) to the *Franz Lisp* environment. The user responds to the first "->" prompt with "(load 'cmufncs)" and to the second with "(load 'cmuenv)". These two load commands set up the *Lisp* environment, loading the library of built-in *Lisp* functions, etc. The system responds with the lines in brackets and the "t." The dskin command loads the given file into the current *Lisp* environment. As a result of the command (dskin validate), program VALIDATE can now be used. Function *interact* initiates an interactive loop:

```
3.(interact)
Enter wff to be validated or nil to terminate session
((A => B) @ (((A ^ B) => C) @ (A => C)))
(The formula ((A => B) @ (((A ^ B) => C) @ (A => C))) is valid.)
Enter wff to be validated or nil to terminate session
(((A => B) ^ (B => A)) @ (A <=> B))
((The formula (((A => B) ^ (B => A)) @ (A <=> B)) is invalid. A false model exists:)
(world value-assignment ((1 (A 1) (B 0) ((A => B) 1) ((B => A) 1)) (2 (A 1) (B 1)) (3 (B 1) (A 1))))
(arrows ((A 2 3) (A 3 2) (A 1 2) (A 1 3))))
Enter wff to be validated or nil to terminate session
nil
Session terminatednil
```

The program responds to the command (interact) with "Enter wff to be validated or nil to terminate session." A wff is entered next for testing. The program responds with the valid message or the invalid message. Note that responding to the prompt "Enter wff to be validated or nil to terminate session" with nil terminates the interactive session.* The user is, however, still in the *Lisp* environment. Command (exit) causes a change from the *Lisp* environment back to the operating system environment. Before exiting,

* The *Lisp* print command prints "nil" following the argument to the print, e.g., "Session terminatednil."

though, function *test* can also be used to validate wffs:

```
4.(test f8)
((The formula (((A => C) ^ (B => C)) @ ((A ^ B) => C)) is invalid. A false model exists:)
(world value-assignment ((1 (A 0) (C 0) (B 0) ((A => C) 1) ((B => C) 1) (((A ^ B) => C) 0))
(-4 (A 1) (B 1) (C 0)) (2 (A 1) (C 1) (B 0)) (3 (B 1) (C 1) (A 0))))
(arrows ((A -4 2) (A -4 3) (A 2 3) (A 3 2) (A 1 -4) (A 1 2) (A 1 3))))nil
5.(test '(((A => B) ^ (~ (A => (~ C)))) @ ((A ^ C) => B)))
(The formula (((A => B) ^ (~ (A => (~ C)))) @ ((A ^ C) => B)) is valid.)nil
6.(exit)
sfu_cmpt%
```

Note that a single quote precedes a formula when function *test* is called as in line 5, but not when the argument to *test* is the name of a formula as in line 4. No quote is used when entering wffs within the *interact* loop. Command (exit) returns the user to the operating system environment.

Another option is to put the *Lisp* commands in a file, then run *Lisp* with the file as input and direct the output to another file. This is accomplished in the following way. Create a file, say *testwffs*, and insert the commands:

```
(load 'cmufncs)
(load 'cmuenv)
(dskin validate)
(dskin examples)
```

Next enter calls to function *test* for each wff to be validated and command (exit) to leave the *Lisp* environment:

```
(test f1)
(test f2)
(test f3)
(test f4)
(test '((((A => B) ^ ((A ^ B) => C) ^ A) @ (A => C)))
(test '((A ^ B) @ (A => B)))
(exit)
```

Then all that has to be done is to type the following line from the operating system environment:

```
%lisp < testwffs > results
```

The results for each wff tested will be in file *results*. File *ex.results* contains editor commands which format the results into a more readable form. Formatting is accomplished by typing the following line from the operating system environment:

%ex results < ex.results

File *results* will then be formatted in a more readable style.

5. Sample I/O

The following is a sample interactive session:

```
sfu_cmpt% lisp
Franz Lisp, Opus 38.91
-> (load 'cmufncs)
[fasl /usr/local/lib/lisp/cmufncs.o]
t
-> (load 'cmuenv)
[load /usr/local/lib/lisp/cmuenv.l]
[fasl /usr/local/lib/lisp/cmumacs.o]
[fasl /usr/local/lib/lisp/cmufncs.o]
[fasl /usr/local/lib/lisp/cmutpl.o]
[fasl /usr/local/lib/lisp/cmufile.o]
t
1.(dskin validate)
[load validate]
(validate)
2.(dskin examples)
[load examples]
(examples)
3.(interact)
Enter wff to be validated or nil to terminate session
((A => B) @ (((A ^ B) => C) @ (A => C)))
(The formula ((A => B) @ (((A ^ B) => C) @ (A => C))) is valid.)
Enter wff to be validated or nil to terminate session
(((A => B) ^ (B => A)) @ (A <=> B))
((The formula (((A => B) ^ (B => A)) @ (A <=> B)) is invalid. A false model exists:)
(world value-assignment ((1 (A 1) (B 0) ((A => B) 1) ((B => A) 1)) (2 (A 1) (B 1)) (3 (B 1) (A 1))))
(arrows ((A 2 3) (A 3 2) (A 1 2) (A 1 3))))
Enter wff to be validated or nil to terminate session
A => A
(The formula (nil nil nil) is invalid. A false model exists: ((A 0)))
Enter wff to be validated or nil to terminate session
Error: NAMESTACK OVERFLOW
<1>: Error: Unbound Variable: A
<2>: (reset)
[Return to top level]
4.(interact)
```

```
Enter wff to be validated or nil to terminate session
(A => A)
(The formula (A => A) is valid.)
Enter wff to be validated or nil to terminate session
nil
Session terminatednil
5.(test f8)
((The formula (((A => C) ^ (B => C)) @ ((A ^ B) => C)) is invalid. A false model exists:)
(world value-assignment ((1 (A 0) (C 0) (B 0) ((A => C) 1) ((B => C) 1) (((A ^ B) => C) 0))
(-4 (A 1) (B 1) (C 0)) (2 (A 1) (C 1) (B 0)) (3 (B 1) (C 1) (A 0))))
(arrows ((A -4 2) (A -4 3) (A 2 3) (A 3 2) (A 1 -4) (A 1 2) (A 1 3))))nil
6.(test f11)
(The formula (((A => C) ^ (B => C)) @ ((A v B) => C)) is valid.)nil
7.(test '((A ^ B) @ (A => B)))
((The formula ((A ^ B) @ (A => B)) is invalid. A false model exists:)
(world value-assignment ((1 (A 1) (B 1) ((A => B) 0)) (-2 (A 1) (B 0))))
(arrows ((A 1 -2))))nil
8.(test '(((A => B) ^ (~ (A => (~ C)))) @ ((A ^ C) => B)))
(The formula (((A => B) ^ (~ (A => (~ C)))) @ ((A ^ C) => B)) is valid.)nil
9.(exit)
sfu_cmpt%
```

Note that:

(1) No quote precedes a wff entered inside the interact loop.

(2) No quote precedes the name of a wff.

(3) A single quote precedes a wff entered as an argument to function *test*.

(4) A wff must be entered in infix notation inside the interact loop and as an argument to function *test*.

(5) The name of a wff cannot be used inside the interact loop but may be used as an argument to function *test*.

(6) When a formula is entered which is not well-formed or is ambiguous, *Lisp* returns an error message. For example, "A => A" is entered in the sample session, and "Error: NAMESTACK OVERFLOW" is returned. Command (reset) returns the user to the top level within the *Lisp* environment. Command (interact) gets the user back into the loop where the wff is entered in its proper form "(A => A)".

(7) Nil ends the loop created by the interact command.

(8) Command (exit) ends the *Lisp* session and returns the user to the operating system environment.

To illustrate the second option, file *testwffs* is created containing:

```
(load 'cmufncs)
(load 'cmuenv)
(dskin validate)
(dskin examples)
(print "formula f1")
(test f1)
(print "formula f2")
(test f2)
(print "formula f3")
(test f3)
(print "formula f6")
(test f6)
(print "formula ((A ^ B) @ (A => B))")
(test '((A ^ B) @ (A => B)))
(print "formula (((A => B) ^ (~ (A => (~ C)))) @ ((A ^ C) => B))")
(test '(((A => B) ^ (~ (A => (~ C)))) @ ((A ^ C) => B)))
(print "done")
(exit)
```

Then the following two lines are typed from the operating system environment:

```
%lisp < testwffs > results

%ex results < ex.results
```

Results now contains:

```
formula f1
The formula (~ ((A => C) ^ ((A ^ B) => (~ C)))) is invalid.
A false model exists:

 world value-assignment
((1 (A 0) (C 0) (B 0) ((A => C) 1) (((A ^ B) => (~ C)) 1))))
 arrows
((1 (~ A)) (1 (~ (A ^ B))))))

formula f2
The formula ((A => B) @ (((A ^ B) => C) @ (A => C))) is valid.

formula f3
The formula (((A => B) ^ (A => C)) @ (A => (B ^ C))) is valid.

formula f6
The formula (((A => B) ^ (B => A)) @ (A <=> B)) is invalid.
A false model exists:

 world value-assignment
((1 (A 1) (B 0) ((A => B) 1) ((B => A) 1))
```

```
(2 (A 1) (B 1))
 (3 (B 1) (A 1))))
 arrows
((A 2 3) (A 3 2) (A 1 2) (A 1 3))))

formula ((A ^ B) @ (A => B))
The formula ((A ^ B) @ (A => B)) is invalid.
A false model exists:

 world value-assignment
((1 (A 1) (B 1) ((A => B) 0))
 (-2 (A 1) (B 0))))
 arrows
((A 1 -2))))

formula (((A => B) ^ (¯ (A => (¯ C)))) @ ((A ^ C) => B))
The formula (((A => B) ^ (¯ (A => (¯ C)))) @ ((A ^ C) => B)) is valid.
```

File *test.data* contains the *Lisp* function calls to test the wffs in file *examples*. To obtain formatted results for these formulas type the two lines:

```
%lisp < test.data > results

%ex results < ex.results
```

PART II. MAINTAINING THE PROGRAM

1. Introduction

The purpose of program VALIDATE is to determine the validity of well-formed formulas (wffs) expressed in conditional logical system N [Delgrande 86]. Since N subsumes propositional logic, the program handles wffs of propositional logic as well. A wff of N is given as input, and VALIDATE returns either a message stating that the given wff is valid or a message stating that the given wff is invalid. If invalid, a set of value assignments to variables is also returned in which the wff evaluates to false.

The algorithm is an implementation of the method of N-diagrams, a tableau-based approach with modifications to handle the possible worlds semantics of logic N. An attempt is made to construct a consistent configuration. If such a configuration can be constructed, the given wff is invalid. Otherwise such a configuration is shown to be impossible to construct, and the wff is therefore valid. The program is written in *Franz Lisp* code and is intended to be run on the *Franz Lisp* interpreter.

2. The Algorithm

Program VALIDATE is an implementation of the method of N-diagrams. A table called *nstruct* is built which represents the semi-complete structure of N-diagrams. Then, for each structural template on the semi-complete structure, RTFC configurations are generated and tested until all RTFC configurations have been tested or a consistent configuration is found. The algorithm for the implementation follows:

Step 1. A wff is entered in one of two ways: (1) interactively within the loop instigated by function interact, or (2) as an argument to function test. Function test calls prefix to convert *wff* to prefix notation. The main operator is assigned 0 when test calls apply-rules sending it arguments *wff* and 0. *Nstruct* is initialized to w_1 where $\neg\omega$ is true.

Step 2. Function apply-rules is called from test to apply propositional rules to *wff*. The application of the rules results in the construction of *valassns*, *mustbe*, and *gen*, data structures from which alternative value assignments to variables and gamma-subformulas are generated. Application of the rule for new

Algorithm NTP

Input: wff

Output: valid, invalid

1. Initialize *nstruct*
   - 1.1. Get *wff*
   - 1.2. Put *wff* in prefix form
   - 1.3. Assign 0 to the main operator of *wff*
2. Build the semi-complete structure(s)
   - 2.1. Repeat
     - Apply α-rules
     - Apply rules for crosses, asterisks, and modified rule for alternatives
     - Apply rule for new worlds

     Until rules applied as often as possible
3. Repeat {for each semi-complete structure}
   - 3.1 Generate structural templates from SCS
   - 3.2. Test templates
   - 3.3. If template is inconsistent then remove template from *paths*
   - 3.4. If *paths* ≠ {} then
     - 3.5 Build *necvals* and *arrows* {tables of forced values and arrow constraints}
     - 3.6 Repeat {for each structural template}
       - $q \leftarrow 0$
       - 3.7. Repeat {for $q = 1$ to n classes in *eqtemp*}
         - $q \leftarrow q + 1$
         - Initialize *eqtemp* for q classes
         - 3.8. Repeat {for each *eqtemp* of q classes}
           - Initialize *config* template from *eqtemp*
           - 3.9 Repeat {for each *config* template}
             - Generate *arwset*
             - Test configuration
             - $d \leftarrow q - 1$
             - Get next *config* template

           Until $d = 0$ {all *config* templates tested} or ccf
           - $c \leftarrow q - 1$
           - Get next *eqtemp* of q classes

         Until $c = 0$ {all *eqtemps* of q classes tested }or ccf

       Until $q = n$ {all *eqtemps* tested} or ccf

     Until all structural templates tested or ccf

     Get next SCS

   Until all alternative SCS tested or ccf

worlds results in the extension of *nstruct* to the SCS once the rules have been applied as often as possible.

Step 3. The second phase of the algorithm generates and test RTFC configurations. If at any point in step 3 a consistent configuration is found, the algorithm terminates. The outer loop is repeated for each SCS as there may be more than one. Structural templates are generated from the current SCS and stored in *paths* (step 3.1). The templates are tested and inconsistent templates removed from *paths* (steps 3.2, 3.3). If at least one template remains in *paths*, then *necvals*, the table of forced values, and *arrows*, the table of arrow constraints, are built (step 3.5).

The loop of step 3.6 is repeated for each structural template remaining in *paths*. Structural templates consisting of more than two worlds are not forward-connected. Therefore an arrow in one direction or the other or both must be added between each pair of worlds in the configuration to make the configuration RTFC. However, all possible ways of adding arrows need not be tested. Transitivity fails for some combinations. Thus only sets of arrows which create RTFC configurations are generated. This type of generation is accomplished in the following way.

The accessibility relations and worlds in a configuration are viewed in terms of a well-ordering on equivalence classes of worlds (steps 3.6-3.9). Reflexivity and transitivity hold, but symmetry is not imposed by the accessibility relation. Thus there may occur from one to n equivalence classes, where n is the number of worlds in the configuration, excluding w_1 (step 3.7). *Eqtemp* is the data structure representing the current equivalence class structure. The ordering of EQ classes also varies (step 3.8). For example, if there are three worlds in a configuration (excluding w_1), then the successive values of *eqtemp* are: (3), (1 2), (2 1), (1 1 1). From each *eqtemp* consisting of more than one equivalence class, there are different ways in which the world labels may be fitted into the equivalence classes (step 3.9). Consider worlds w_2, w_3, w_4 and *eqtemp* (1 1 1). There are six ways of fitting three worlds labels into three equivalence classes: (2 3 4), (2 4 3), (3 2 4), (3 4 2), (4 2 3), (4 3 2). *Config* is the data structure representing the current permutation of world labels from *eqtemp*. *Arwset* is the data structure representing the set of arrows to be added to a structural template to form an RTFC config. It is generated from *config* and tested for consistency. If consistent, a false model has been found. *Arwset*, *path*, and *wrldvals*, the set of value assignments to relevant variables and gamma-subformulas at each world in the configuration, are returned. These data structures describe the false model. If inconsistent, the next RTFC configuration is generated and tested. This cycle continues until all RTFC configurations have been tested or a consistent configuration found.

3. Function Types

The functions of program VALIDATE can be categorized on the basis of the type of work performed. There are three main types of functions: (1) those which build or modify the data structures, (2) those which perform tests, and (3) those which manipulate lists. Section 3.1 provides a description of the

main data structures of program VALIDATE and lists the functions which initialize, build, or modify them. Section 3.2 presents the functions which perform tests, whereas the list manipulation functions are presented in section 3.3.

3.1. Data Structures

The present section concerns the functions which initialize, build, or modify data structures. The functions are grouped according to the data structures they manipulate. The section is comprised of descriptions of the main data structures of program VALIDATE and lists of the functions which manipulate them. The documentation for each data structure consists of four parts:

(1) a header comprised of the name of the data structure in bold,

(2) a short description of the structure,

(3) an example, and

(4) a list of the functions responsible for initializing, building, or modifying the data structure.

The data structures appear in the order in which they arise within the program.

wff

The formula to be tested for N-validity is entered in infix notation in one of two ways. It is either entered at the terminal and read through functions interact and readexpr or entered as an argument to function test. Functions prefix and prefixw convert *wff* to prefix notation, whereas function inorder converts a wff in prefix form to infix form. A wff in infix form is in the order *operand-operator-operand*, or *operator-operand* if the operation is negation. A wff in prefix form is in the order *operator-left operand-right operand*, or *operator-operand* if the operation is negation. For example, given $(((A \Rightarrow B) \wedge (A \Rightarrow C)) @ (A \Rightarrow (B \wedge C)))$ in infix notation, its prefix form is $(@ (\wedge (\Rightarrow A B)(\Rightarrow A C))(\Rightarrow A (\wedge B C)))$.

valassns, mustbe

Function apply-rules assigns 0 to the main operator of *wff* and applies the rules of propositional logic for the assignment of truth values to variables and to gamma-subformula. *Mustbe* is the data structure containing top level alpha assignments to variables and gamma-subformula, whereas *valassns* contains a list of alternative value assignments as in the case of a true logical ∨ (OR). For example, from *wff* $(((A \Rightarrow B) \wedge (B \Rightarrow A)) @ (A \Leftrightarrow B))$ we get:

mustbe: ((⇒ A B 1) (⇒ B A 1))

valassns: (((A 1) (B 0)) ((A 0) (B 1))).

The functions which build or modify *valassns* and *mustbe* are apply-rules, apply-betarule, avalues, bvalues, alpha, beta, gamma, init-mustbe.

gen

Gen is the generator by which the next alternative set of value assignments can be obtained from *valassns* and *mustbe*. The generator consists of a list of numbers, 0-2, the length of which is equal to the maximum number of ORs in *valassns*. For example, given:

mustbe: ((A 0))

valassns: (((B 1) (C 0)) ((B 0) (C 1)) ((B 0) (C 0))),

gen is initialized to (-1). When *gen* is (0) the first set of value assignments is ((*A* 0) (*B* 1) (*C* 0)); for *gen* (1) the set of value assignments is ((*A* 0) (*B* 0) (*C* 1)); for *gen* (2) ((*A* 0) (*B* 0) (*C* 0)). The functions which build or modify *gen* are init-gen, depth, maxnum, buildgen, update-gen, next-assn.

nvals, nmust, ngen

These data structures have the same form as *valassns*, *mustbe*, and *gen* but are used in the case that a gamma-subformula occurs within the scope of a beta-subformula. When this case arises there is more than one alternative *nstruct* to test. The functions which build or modify *nvals*, *nmust*, and *ngen* are init-nmust, apply-rules, apply-betarule, avalues, bvalues, alpha, beta, gamma, init-gen, depth, maxnum, buildgen,

update-gen, next-nassn.

nstruct

Nstruct is the main data structure in the program. It is a table representing a system of N-diagrams. Given the system of Figure 2,

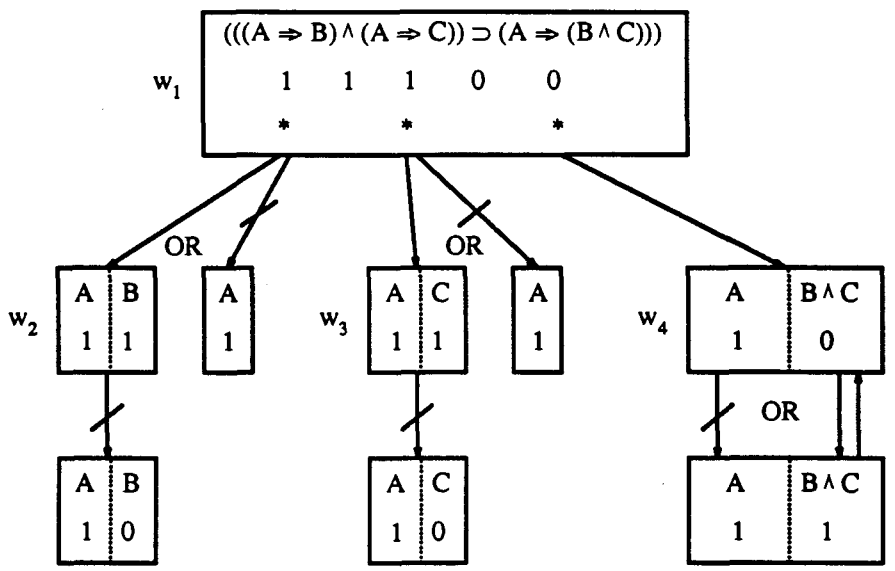


Figure 2. Semi-complete system of N-diagrams for formula $(((A \Rightarrow B) \wedge (A \Rightarrow C)) @ (A \Rightarrow (B \wedge C)))$.

nstruct is:

```
((1 (~ wff) (^ (^ (V (A 1 2) (~ A)) (V (A 1 3) (~ A))) (A 1 -4)))
 (2 (^ A B) (V (~ A) B))
 (3 (^ A C) (V (~ A) C))
 (-4 (^ A (~ (^ B C))) (V (V (~ A) (~ (^ B C))) (A wprime -4))))
```

The numbers in the first column of the table are world labels. Note that a world created from a false ⇒ is given a negative label. This makes it easy to distinguish these worlds from worlds created due to a true ⇒ when it is necessary to do so, e.g., in building arrow constraints. The second column consists of the relevant true conditions at a world. This is the information inside the rectangles of the diagrams. So for example, A and B are both true at world w_2 above.

The third column consists of the conditions which must hold at worlds accessible to the current world. Consider row 1 of the example *nstruct*. The third column contains arrows (A 1 2), (A 1 3), (A 1 -4). The conditions on accessibility from world w_1 are that:

(1) either there is an accessible world w_2 or A is false at all accessible worlds,

(2) either there is an accessible world w_3 or A is false at all accessible worlds, and

(3) there must be an accessible world w_{-4}.

At worlds accessible to world w_3 it must be the case that either A is false or C is true. (It cannot be the case that A is true and C false.) At worlds accessible to world w_{-4} (created from a false $\Rightarrow$ at w_1) either (1) A is false or ($B \wedge C$) is true (it cannot be the case that A and ($B \wedge C$) are both true), or (2) if A and ($B \wedge C$) are both true, then there must be an arrow back to world w_{-4}. The functions which build or modify *nstruct* are init-nstruct, build, gamma-true, gamma-false, modifiedt, modifiedf, newrowt, newrowf, update-nstruct, update-worlds.

arrows, nested-arws

Arrows is a data structure containing information about transitive arrows, inconsistent arrows, arrows which cannot co-occur, arrows which must co-occur, nested-arrows, and betas in arrows and/or worlds. An example structure *Arrows* follows:

```
((+ (A 2 4))
 (~ (A 2 3))
 (~ (A 4 2) (A 3 2))
 (-> ((A -5 2) (A 3 2)) (A 2 -5))
 (-> ((A -5 3) (A -6 3)) (∨ (A 3 -5) (A 3 -6)))
 (@ (A 2 4) ((A 1 4)))
 (* -6)
 (* (A 4 -5))
 (* (1 (~ (^ A B)))) )
```

Row (+ (*A* 2 4)) means that arrow (*A* 2 4) must occur in any configuration in which worlds w_2 and w_4 are both present. Row (~ (*A* 2 3)) indicates that arrow (*A* 2 3) is inconsistent and therefore cannot occur in any consistent configuration. Row (~ (*A* 4 2) (*A* 3 2)) indicates that arrows (*A* 4 2) and (*A* 3 2) cannot co-occur in any consistent configuration. Row ($\rightarrow$ ((*A* −5 2) (*A* 3 2)) (*A* 2 −5)) means that if arrows (*A* −5 2) and

(*A* 3 2) co-occur, then (*A* 2 –5) must also occur. This case arises when arrow (*A* 3 2) forces value assignments at world w_2 which in turn makes (*A* –5 2) inconsistent unless (*A* 2 –5) also occurs. Similarly row (→ ((*A* –5 3) (*A* –6 3)) (∨ (*A* 3 –5) (*A* 3 –6))) indicates that if arrows (*A* –5 3) and (*A* –6 3) co-occur, then either (*A* 3 –5) or (*A* 3 –6) must also occur in the configuration. Row (@ (*A* 2 4) ((*A* 1 4))) represents transitive arrows. If arrow (*A* 2 4) occurs in a configuration then arrow (*A* 1 4) must also occur since accessibility is transitive.

An asterisk indicates the occurrence of a beta-subformula. Row (* –6) means that there exists a beta-subformula within the conditions at world w_{-6}. Row (* (*A* 4 –5)) means that a beta-subformula occurs within either the conditions for accessibility from w_4 or the conditions at w_{-5}. Thus the arrow of *necvals* (*A* 4 –5) contains a beta-subformula. Row (* (1 (~ (^ *A B*)))) indicates that not-arrow (1 (~ (^ *A B*))) contains a beta-subformula. The functions which build or modify *arrows* are updated-arws, wkworlds, wkworld, gsum, update-arws, trans, get-trans, modfarws, update-arrows, double-arw, ifwiwj, ifwi, updated-arrows, cant-co-occur, cannot, cant, check-pairs, upd-arw, betanots, add-arws, beta-nestednecs.

Nested-arws has the same form as *arrows* but handles nested-arrows and their return-arrows. Once *nested-arws* is built, it is appended to *arrows*. The functions which build or modify *nested-arws* are mustarws, checkarws, update-nestarws, checkwks, cons-wks, upd-nestarws.

necvals, nested-necs

Necvals is a data structure consisting of the set of arrows to be added to a configuration to make it RTFC. Associated with each arrow is (1) the set of value assignments which must hold in order that the arrow be consistent, and (2) the term "cons" or "incons" to indicate the consistency of the arrow, for example:

(((A 3 2) (A 1) (B 1) (C 1) cons)
((A -4 2) (A 1) (B 1) (C 0) cons)
((A 2 3) (A 1) (B 1) (C 1) cons)
((A -4 3) (A) (B) (C) cons (A 3 -4))
((A 2 -4) (A) (B) (C) incons)
((A 3 -4) (A 1) (B 0) (C 0) cons)).

From the example *necvals* we see that arrows (*A* 3 2), (*A* –4 2), (*A* 2 3), and (*A* 3 –4) are all consistent. Arrow (*A* –4 3) is consistent as long as arrow (*A* 3 –4) also occurs in any configuration in which (*A* –4 3) occurs. Arrow (*A* 2 –4) is inconsistent. This information is used to build *arrows*. The functions which build or modify *necvals* are test-arrows, update-necvals, check-wprime, add-vals, add-val, rtnarws, awjwi, init-necvals, generate-test.

Nested-necs has the same form as *necvals* but handles nested-arrows and their return-arrows. Once *nested-necs* is built, it is appended to *necvals*. The functions which build or modify *nested-necs* are update-nested-necs, doublearew-incons, nestedarw-incons, test-rtnarw.

svals, smust, sgen

These data structures have the same form as *valassns*, *mustbe*, and *gen* but are used in testing arrow and not-arrow consistency. The functions which build or modify *svals*, *smust*, and *sgen* are incons-smust, apply-rules, apply-betarule, avalues, bvalues, alpha, beta, gamma, init-gen, depth, maxnum, buildgen, update-gen, next-sassn, nsat, add-arw.

paths

Paths is a data structure which represents all possible alternative structural templates along the semi-complete structure of N-diagrams. Consider the system of N-diagrams of Figure 2. *Paths* for this semi-complete structure is:

(((A 1 -4) (1 (~ A)) (1 (~ A)))
((A 1 -4) (1 (~ A)) (A 1 3))
((A 1 -4) (A 1 2) (1 (~ A)))
((A 1 -4) (A 1 2) (A 1 3))).

The first path represents a configuration in which two worlds exist, w_1 and w_{-4}. There are two not-arrows both indicating that *A* must be false at all worlds accessible from w_1. The last path represents a semi-complete structure in which four worlds exist, w_1, w_2, w_3, and w_{-4}. There are no not-arrows on this path.

The functions which build or modify *paths* are init-paths, transitive, intrans, complete, comp, newnrows, nested-paths, expanded, newrows, test-template, test-temp, add1base2.

worldvals, wrldvals

Data structure *worldvals* contains value assignments to variables and gamma-subformula which must hold at each world. It is initialized to the conditions which hold at each world from *nstruct*. Consider an example:

((1 (A 1) (B 1) (C 0) (⇒ A B 1) (⇒ C B 0))
(-3 (C 1) (B 0) (A 0))
(2 (A 1) (B 1) (C 0))).

At world w_1 A and B are true, C false, $A \Rightarrow B$ true, and $C \Rightarrow B$ false. At world w_{-3} A and B are both false and C true, whereas at w_2 A and B are both true and C false. The functions which build or modify *worldvals* are init-worldvals, update-wvals, wigamma, upd-worldvals.

Wrldvals has the same form as *worldvals*. In fact it is initialized to *worldvals*. Once an arrow set has been determined to form an RTFC configuration, the values which must hold in order for the added arrows to be consistent (from *necvals*) are added to *wrldvals*. If that arrow set is found to be inconsistent, then *wrldvals* is re-initialized to *worldvals* and the next arrow set generated and tested. If, on the other hand, the configuration is found to be consistent, *wrldvals*, *arwset*, and *path* are returned as a false model. The functions which build or modify *wrldvals* are upd-wrldvals, update-newvals.

eqtemp

Data structure *eqtemp* is the equivalence class template from which a configuration template (and subsequently an *arwset*) is generated. *Eqtemp* is initialized to one equivalence class containing all worlds in the configuration (excluding w_1). Once all possible configurations have been generated, *eqtemp* is incremented. For example, if there are three worlds (excluding w_1) in the current semi-complete structure, the successive values of *eqtemp* are:

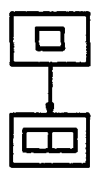
(1) (3)

(2) (1 2)

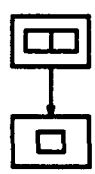
(3) (2 1)

(4) (1 1 1)

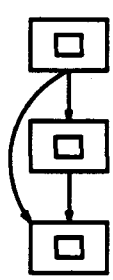
Eqtemp (1) consists of one equivalence class containing all three worlds. *Eqtemp* (2) consists of two equivalence classes, the first containing one world, the second two worlds:



Eqtemp (3) also consists of two equivalence classes, the first containing two worlds and the second one world:



Eqtemp (4) consists of three equivalence classes, one world in each:



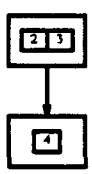
The functions which build or modify *eqtemp* are init-eqtemp, next-eqtemp.

config

Data structure *config* is the current configuration of worlds (generated from *eqtemp*) from which the set of added arrows is generated. So if the current *eqtemp* is (2 1) and worlds on the current path are w_2, w_3, and w_4, then *config* has the following successive values:

((2 3) (4))
((2 4) (3))
((3 4) (2))

The values of *config* represent all possible ways in which the world labels can be fitted into the current *eqtemp*. The first *config* above represents a configuration comprised of two equivalence classes, the first containing worlds w_2 and w_3, the second containing world w_4:



The functions which build or modify *config* are init-config, init-class, next-config, next-world.

arwset

Data structure *arwset* is the set of arrows added to the semi-complete structure to form an RTFC configuration. It is generated from *config* in the following way:

Algorithm Arrowset

1. *arrowset* ← {}
2. Repeat
   2.1 For each w_i in topmost EQ class do
      (a) For each w_j, $i \neq j$, in topmost and lower EQ classes do
         $arrowset \leftarrow arrowset \cup \{Aw_iw_j\}$
   2.2 Remove topmost EQ class

   Until no EQ classes left

Thus the set of arrows generated from *config* ((2 3) (4) (5 6)) is {(Aw_2w_3), (Aw_3w_2), (Aw_5w_6), (Aw_6w_5), (Aw_2w_4), (Aw_2w_5), (Aw_2w_6), (Aw_3w_4), (Aw_3w_5), (Aw_3w_6), (Aw_4w_5), (Aw_4w_6)}. The functions which build or modify *arwset* are get-arwset, update-arwset, c1, ci, upd-arwset.

accessbl

Data structure *accessbl* contains gamma-subformulas from *wrldvals* which can be verified, i.e., for which accessible worlds w_j exist making the value assigned gamma at w_i hold. These subformula are then removed from *wrldvals*. This is done so that a gamma which can be verified before trying alternatives does not have to be re-verified for each alternative value assignment from *alterns*. Consider an example:

((1 (⇒ (⇒ A B) (⇒ B A) 0))
(-2 (⇒ A B 1) (⇒ B A 0))
(-4 (⇒ B A 0))).

The gamma ($(A \Rightarrow B) \Rightarrow (B \Rightarrow A)$) is verified to be false at w_1 since ($A \Rightarrow B$) is true and ($B \Rightarrow A$) false at w_{-2}. The form of *accessbl* is the same as that of *worldvals* and *wrldvals*. The functions which build or modify *accessbl* are build-accessbl, get-accessbl.

alterns, altern

Data structure *alterns* is a list of alternative value assignments to propositional variables at each world in the configuration. Variable assignments which must hold (from *wrldvals*) are appended to the alternatives for the remaining unassigned variables. This structure is designed to handle situations in which application of the rule for alternatives was delayed. In some cases arrow(s) force assignments to β_1, β_2, but in others they do not. *Atlerns* provides the alternative assignments without generating all the alternative SCS. Consider an example:

((1 ((B 1) (C 1) (A 0))
((B 1) (C 1) (A 1)))
(2 ((A 1) (B 1) (C 0)))
(-3 ((A 1) (C 0) (B 1)))).

B and *C* must both be true at world w_1, but *A* may be true or false. There is only one alternative value assignment to variables at worlds w_2 and w_{-3} above. The functions which build or modify *alterns* are get-alternatives, get-alterns, get-alts, upd-alterns, add1base2.

Data structure *altern* is initialized to *alterns*. To get an alternative list of value assignments to variables at each world in the configuration, the first element of the set of alternatives at each world is returned. *Altern* is then updated by removing an alternative element. The next time *altern* is accessed a new set of alternatives is returned. The successive values of *altern* for the example *alterns* above are:

```
(1) ((1  ((B 1) (C 1) (A 0))
         ((B 1) (C 1) (A 1)))
     (2  ((A 1) (B 1) (C 0)))
     (-3 ((A 1) (C 0) (B 1))) )

(2) ((1  ((B 1) (C 1) (A 1)))
     (2  ((A 1) (B 1) (C 0)))
     (-3 ((A 1) (C 0) (B 1))) ).
```

The alternatives generated from these values are:

```
(1) ((1  (B 1) (C 1) (A 0))
     (2  (A 1) (B 1) (C 0))
     (-3 (A 1) (C 0) (B 1)) )

(2) ((1  (B 1) (C 1) (A 1))
     (2  (A 1) (B 1) (C 0))
     (-3 (A 1) (C 0) (B 1)) ).
```

The functions which build or modify *altern* are next-altern, update-altern.

3.2. Tests

There are three main types of functions which perform tests: (1) type tests, (2) consistency tests, and (3) false-configuration tests. Type and consistency tests perform a test, then return t if the test is passed and nil otherwise. Type testing determines whether the given argument is of a certain type, or if elements of a certain type occur within a list argument. Consistency checking determines whether or not the given

argument(s) are inconsistent. An argument may be a subformula, an arrow, a set of value assignments, a not-arrow and the conditions at a world, etc. False-configuration testing performs a consistency check on a configuration, then returns the configuration if consistent, and nil otherwise. The following lists identify these functions:

Consistency checking: altassns, applies, b=>sat, case1beta, case2beta, case3beta, check-necvals, check-nested, consistent, consistentn, determ, doublearw-incons, elim~path, foundwj, incons, incons+, incons~, incons->, incons-arwset, incons-beta, incons=>beta, incons-betaconds, incons-betanecs, incons-betanots, incons-gamma, incons-necvals, incons-notarrows, incons-notarws, incons-smust, incons-winot-arws, incons-winots, inconsistent, nestedarw-incons, no-access, no-accesswk, nsat, orfalse=>, or=>fal se, reflex1, reflexive, reflex-test, reflexw1, test-betas, test-config, test-notarrows, test-notarws, test-rtnarw, test-semicomp, verify, verify=>, verif=>0, verify=>0, verif=>1, verify=>1, verify-false=>, verify-true=>, verify-wval=>, verify-wvals=>.

Type testing: alpha, alphascope, altlist, arrowp, beta, diffv, diffval, diffvals, gamma, in, in-any, inarw, nested, nested-arrow, nested-gammas, notarrowp, op, ored, orpath, propvar, propwff, top-level-var, varassn.

False-configuration testing: generate-test, gen&test-configs, test-path, test-paths, validate, validnest, verify-falseconfig, verify-flsconfig, verify-fconfig.

3.3. List Manipulation

The functions in the list manipulation category modify lists in one of three ways. The "get" functions get a list of elements of a certain type from the list argument to the function, or find a certain element in a list. The "remove" functions remove certain elements from a list argument to the function. The elements to be removed are usually given as an argument as well as the list from which the elements are to be removed. The "change form" functions alter the form of the arguments. For example, function inorder converts a wff in prefix form to infix form. The following lists identify these functions:

Get: accesswj, access-worlds, allpaths, back, checkdarw, count, element-of, elemn, find, findlist, findn, findrow, findworlds, front, get-alternatives, get-alterns, get-alts, get-ands, get-arws, get-betas, get-conds,

get-fconfig, getfmodel, get-gamma, get-genlist, get-mustbe, get-notarws, get-path, get-valist, get-vals, get-varassns, get-vars, get-wiarws, get-winots, get-worldlist, get-worlds, get-wrlds, infix-assns, infix-gammas, infix-notarws, intersect, intersection, patharws, sumtop, top, update -alterns, wjconds, wjs, wkset.

Remove: check-paths, del, delet, delet-ands, elim-elem, remv-access, remvdup, remvdups, remv-incons, remvlist, remv-mustvals, remv-nested, remvnull, remv-semiarws, remvar, remv-wiarws, remvwinots, remvwkset, wiconds.

Change form: andform, andwff, converted, flat, inorder, merge, prefix, prefixw, subf-rules, subval, transform, valsubf.

4. Function Descriptions

This section consists of descriptions of the functions of program VALIDATE. The documentation for each function is comprised of four parts:

(1) a header which includes the function name and arguments to the function,

(2) a short description of what the function does,

(3) a list of other user-defined functions the current function calls, marked by "Calls:" in bold, and

(4) a list of other user-defined functions which call the current function, marked by "Called from:" in bold.

The functions appear in the present section in the same order in which they occur in the program.

(interact)

Function INTERACT enables wffs to be entered and tested interactively. The function calls READEXPR to read the input wff. TEST is called to validate the wff. The interactive session is terminated when the wff read is NIL.

Calls: readexpr, test.

Called from: terminal to instigate an interactive session.

(readexpr)

Function READEXPR prompts the user for an input wff and reads the wff.

Calls: no other user-defined functions.

Called from: interact.

(test nwff)

Function TEST initializes variables and calls VALIDATE to test a given wff. The function returns "valid" if the formula is valid, and a false model otherwise.

Calls: inconsistent, prefix, apply-rules, init-mustbe, init-gen, inorder, validate, first-assn.

Called from: interact, or from terminal to test a wff.

(validate nwff vassn must fmodel)

Function VALIDATE calls NEXT-ASSN to get a new list of value assignments. The alternatives are tested for consistency until one is found to be false, or all alternatives have been tested. The function returns FMODEL, a falsifying model, if the given wff is invalid, and NIL otherwise.

Calls: consistent, consistentn, inorder, propwff, getfmodel, next-assn, nested-gammas, build, init-nstruct, nested, generate-test.

Called from: test.

(inorder nwff)

Function INORDER converts a wff in prefix notation back to left-operand-operator- right-operand form. The function returns the converted wff.

Calls: propvar.

Called from: test, validate, get-fconfig, infix-notarws, infix-gammas.

(consistent assns)

Function CONSISTENT returns T if VARVALS, a list of propositional variables and the values assigned to them, is consistent. A value assignment is consistent if all assignments to a particular variable are the same value. The function returns NIL if any variable has been assigned both 0 and 1.

Calls: inconsistent.

Called from: validate, nsat, verify-true=>, verify-false=>, no-access, accesswj.

(consistentn false-model)

Function CONSISTENTN returns T if function GENERATE-TEST returns a falsifying model, and NIL otherwise.

Calls: no other user-defined functions.

Called from: validnest, test-paths.

(inconsistent vals)

Function INCONSISTENT returns T if the given assignment of values to variables is inconsistent, and NIL otherwise. A value assignment is inconsistent if both 0 and 1 are assigned to the same variable or subformula.

Calls: incons-gamma, get-gamma, incons, remvlist.

Called from: test, consistent, test-rtnarw, add-arw, reflex1, incons-smust, check-necvals, test-notarws, incons=>beta, b=>sat.

(incons-gamma gammas)

Function INCONS-GAMMA checks the value assignments to gamma-subformula for consistency. The function returns T if an inconsistency is found, and NIL otherwise.

Calls: in, front.

Called from: inconsistent.

(get-gamma vals gammas)

Function GET-GAMMA returns a list of the gamma-subformula in the given list VALS.

Calls: no other user-defined functions.

Called from: validnest, inconsistent, nested.

(incons vals)

Function INCONS returns T if the given assigments of values to variables is inconsistent, and NIL otherwise. A value assignment is inconsistent if both 0 and 1 are assigned to the same variable or subformula.

Calls: in, findlist.

Called from: inconsistent.

(getfmodel assns newlist)

Function GETFMODEL returns the given list ASSNS from which duplicates have been removed.

Calls: no other user-defined functions.

Called from: validate, generate-test, update-wvals, nestedarw-incons, test-rtnarw, test-config, get-fconfig, merge, b=>sat.

(apply-rules subf val)

Function APPLY-RULES applies alpha-, beta-, gamma-rules and returns an appended list of value assignments to propositional variables.

Calls: propvar, alpha, avalues, beta, apply-betarule, gamma.

Called from: test, apply-betarule, alphascope, incons-smust.

(apply-betarule subf val)

Function APPLY-BETARULE calls APPLY-RULES to get each alternative assignment for a beta-subformula. The function returns a list of the alternative assignments.

Calls: apply-rules, bvalues.

Called from: apply-rules.

(propvar subf)

Function PROPVAR returns T if subf is a propositional variable, and NIL otherwise.

Calls: in.

Called from: apply-rules, prefixw, inorder, get-vars, valsubf.

(propwff wff)

Function PROPWFF returns T if the given formula is propositional, and NIL otherwise.

Calls: in.

Called from: validate.

(prefix wff)

Function PREFIX returns a copy of the given wff in prefix form.

Calls: propvar.

Called from: test.

(prefixw wff)

Function PREFIXW returns a listed copy of the given wff in prefix form.

Calls: propvar.

Called from: prefix.

(init-mustbe varvalues varval mustbe)

Function INIT-MUSTBE initializes MUSTBE to variable assignments which must be in order that the given wff be false. These elements are removed from VARVALS. The function returns initialized MUSTBE.

Calls: varassn, front, back.

Called from: test.

(varassn v)

Function VARASSN returns T if V is an assignment to a variable or a gamma-subformula, and NIL otherwise.

Calls: no other user-defined functions.

Called from: init-mustbe, top-level-var, altlist, get-mustbe, remvar, init-nmust.

(init-gen vassn)

Function INIT-GEN initializes GEN to a list of 0's with last element equal to -1. The length of the list is equal to the maximum number of OR's in VASSN.

Calls: ored, depth, buildgen.

Called from: test, nested, incons-smust.

(depth vassn ands dep)

Function DEPTH finds the maximum-length path of OR's in the given list VASSN.

Calls: top-level-var, remvar, altlist, maxnum.

Called from: init-gen.

(maxnum numlist n)

Function MAXNUM returns the greatest element in a list of positive numbers.

Calls: no other user-defined functions.

Called from: depth.

(buildgen len gen)

Function BUILDGEN returns a list of length LEN consisting of 0's with the last element equal to -1.

Calls: no other user-defined functions.

Called from: init-gen.

(first-assn vassns must)

Function FIRST-ASSN gets the first assignment from the value assignments VASSNS, the top-level variable assignments MUST, and the generator GEN. The function returns MUST if VASSNS is null. Otherwise function NEXT-ASSN is called to get a possible false model.

Calls: next-assn.

Called from: test.

(next-assn vassn must)

Function NEXT-ASSN gets a list of value assignments from VASSN, MUST, and GEN to be tested for consistency. The function returns a list of top-level value assign- ments to variables.

Calls: add1base3, in, get-assn.

Called from: validate, first-assn.

(get-assn vassn generator)

Function GET-ASSN gets a new set of value assignments from VASSN and GENERATOR. The function returns a list of top-level value assignments to variables.

Calls: ored, buildassn.

Called from: next-assn, next-nassn, next-sassn.

(buildassn vassn ands)

Function BUILDASSN returns a new set of top-level value assignments to variables.

Calls: top-level-var, get-mustbe, remvar.

Called from: get-assn.

(update-gen genr gen1 gen2)

Function UPDATE-GEN SETQ's GEN in the case that a beta with less than three alternative sets of value assignments is found. The effect is that all null sets of value assignments are skipped over. The function returns updated GEN.

Calls: no other user-defined functions.

Called from: buildassn.

(add1base3 oldlist newlist)

Function ADD1BASE3 increments the given list of numbers OLDLIST with base 3 addition. The function returns the incremented list of numbers.

Calls: no other user-defined functions.

Called from: next-assn, next-nassn, next-sassn.

(add1base2 oldlist newlist)

Function ADD1BASE2 increments the given list of numbers OLDLIST with binary addition. The function returns the incremented list of numbers.

Calls: no other user-defined functions.

Called from: allpaths, get-alts.

(ored olist)

Function ORED returns T if the given list OLIST is a list of elements to be ORed, and NIL otherwise.

Calls: top-level-var.

Called from: init-gen, get-assn.

(altlist vals)

Function ALTLIST returns T if the given list VALS consists of sublists of alternative value assignments to variables, and NIL otherwise.

Calls: varassn.

Called from: depth, remvdups, count.

(top-level-var vals)

Function TOP-LEVEL-VAR returns T if VALS contains a variable assigment (as distinct from a list of alternatives) at the top level, and NIL otherwise.

Calls: varassn.

Called from: depth, buildassn, ored.

(get-mustbe must varvalues)

Function GET-MUSTBE finds variable assignments which must be in order that the given subformula be an alternative. The difference between GET-MUSTBE and function INIT-MUSTBE is that there is no setq on VARVALS. The function returns a list of the variable assignments which must be within the given sub-formula.

Calls: varassn.

Called from: buildassn, incons-smust.

(remvar varvalues newv)

Function REMVAR returns a copy of VARVALUES from which top-level variable assignments have been removed.

Calls: varassn.

Called from: depth, buildassn, incons-smust.

(alpha subf val)

Function ALPHA returns T if SUBF is an alpha-subformula, and NIL otherwise.

Calls: no other user-defined functions.

Called from: apply-rules.

(beta subf val)

Function BETA returns T if SUBF is a beta-subformula, and NIL otherwise.

Calls: no other user-defined functions.

Called from: apply-rules.

(gamma subf)

Function GAMMA returns T if SUBF is a gamma-subformula, and NIL otherwise.

Calls: no other user-defined functions.

Called from: apply-rules.

(avalues op val)

Function AVALUES returns a list consisting of the values to be assigned the left-hand and right-hand operands respectively. AVALUES is called when it is known that the SUBF is an alpha-subformula.

Calls: no other user-defined functions.

Called from: apply-rules.

(bvalues op val)

Function BVALUES returns a list consisting of alternative value assignments to a beta-subformula. Each sublist consists of the values to be assigned the left- and right-hand operands respectively.

Calls: no other user-defined functions.

Called from: apply-rules.

(init-nstruct nwff)

Function INIT-NSTRUCT returns table NSTRUCT initialized to the information for world 1.

Calls: no other user-defined functions.

Called from: validate.

(build nstruct vals w)

Function BUILD builds the N-structure from the current set of value assignments FMODEL. The function returns NSTRUCT, a table of worlds, the conditions which hold at each world, and the conditions on accessibility from each world, updated to reflect the occurrence of true and false => operators.

Calls: in, gamma-true, gamma-false.

Called from: validate, update-nstruct, buildnested.

(gamma-true w ante conseq nstruct)

Function GAMMA-TRUE updates NSTRUCT to reflect a true =>. A new world is created. The conditions for accessibility of the world from which the new world is accessible are updated. The function returns updated NSTRUCT.

Calls: modifiedt, findrow, newrowt.

Called from: build.

(gamma-false w ante conseq nstruct)

Function GAMMA-FALSE updates NSTRUCT to reflect a false =>. A new world is created. The conditions for accessibility of the world from which the new world is accessible are updated. The function returns updated NSTRUCT.

Calls: modifiedf, findrow, newrowf.

Called from: build.

(modifiedt row wno new ante)

Function MODIFIEDT adds to the world from which a new world is accessible, the path to that new world created for a true =>. The function returns updated NSTRUCT.

Calls: front, back.

Called from: gamma-true.

(modifiedf row wno new)

Function MODIFIEDF adds to the world from which a new world is accessible, the path to that new world created for a false =>. The function returns updated NSTRUCT.

Calls: front, back.

Called from: gamma-false.

(newrowt new ante conseq)

Function NEWROWT creates a new world and sets up the conditions which hold at the new world and the conditions for worlds accessible from the new world. The function returns the newly created row built for a true =>.

Calls: no other user-defined functions.

Called from: gamma-true.

(newrowf new ante conseq)

Function NEWROWF creates a new world and sets up the conditions which hold at the new world and the conditions for worlds accessible from the new world. The function returns the newly created row built for a false =>.

Calls: no other user-defined functions.

Called from: gamma-false.

(nested-gammas nstruct)

Function NESTED-GAMMAS returns T if NSTRUCT contains nested => operators, and NIL otherwise.

Calls: in.

Called from: validate, init-paths.

(nested nstructr)

Function NESTED handles the case in which a => operator occurs nested within the scope of another => operator. The function calls GENERATE-TEST if no => operator occurs within the scope of a beta operator, and VALIDNEST otherwise.

Calls: buildbs, generate-test, validnest, remvdups, get-vals, init-gen, get-worlds, init-worlds, get-gamma, next-nassn.

Called from: validate.

(validnest betas worlds fmodel)

Function VALIDNEST is called in the case that nested => operators occur within the scope of beta operators. GENERATE-TEST is called with each alternative NSTRUCT until all alternatives have been tried or a false model is found. The function returns a false model if one is found, and NIL otherwise.

Calls: consistentn, generate-test, update-nstruct, updated-arws, get-gamma, next-nassn.

Called from: nested.

(update-nstruct nstruct vals worlds)

Function UPDATE-NSTRUCT calls BUILD to update NSTRUCT using WORLDS to determine the world from which newly created worlds are accessible. The function returns updated NSTRUCT.

Calls: build, update-worlds.

Called from: validnest.

(update-worlds worlds)

Function UPDATE-WORLDS decrements the number of => operators at a world after a new world has been created and NSTRUCT updated. The function returns updated WORLDS.

Calls: no other user-defined functions.

Called from: update-nstruct.

(updated-arws arws nstruct worlds)

Function UPDATED-ARWS adds transitive arrows to ARWS in the case that nested => operators occur within the scope of a beta operator. The function returns updated ARWS.

Calls: update-arws, wkworlds.

Called from: validnest.

(wkworlds worlds nstruct)

Function WKWORLDS returns a list of new worlds created which are accessible from a world in which nested => operators occur.

Calls: wkworld, gsum.

Called from: updated-arws.

(wkworld world nstruct totalw wks)

Function WKWORLD returns a list of new worlds created accessible from a world in which nested => operators occur.

Calls: no other user-defined functions.

Called from: wkworlds.

(gsum worlds gammasum)

Function GSUM returns the total number of nested => operators in WORLDS.

Calls: no other user-defined functions.

Called from: wkworlds.

(remvdups betas newlist)

Function REMVDUPS removes duplicate value assignments to => operators from BETAS.

Calls: altlist, in, remvdup.

Called from: nested.

(remvdup alist newlist)

Function REMVDUP removes a duplicate value assignment to => operators from an alternative list.

Calls: in.

Called from: remvdups.

(get-vals betas vals)

Function GET-VALS returns a list of the alternative values for => operators nested within the scope of beta operators.

Calls: no other user-defined functions.

Called from: nested.

(get-worlds betascope worlds)

Function GET-WORLDS returns a list of the worlds in which nested => operators occur, and the number of nested => operators occurring at each world.

Calls: count, in.

Called from: nested.

(count betas world worlds)

Function COUNT counts the number of nested gammas at a world. The function returns updated WORLDS.

Calls: altlist, front, back, find, in.

Called from: get-worlds.

(init-worlds betascope worlds)

Function INIT-WORLDS initializes WORLDS to a list of sublists consisting of each world in which nested => operators occur and a 0. The function returns initialized WORLDS. sp **Calls:** no other user-defined functions.

Called from: nested.

(next-nassn betas)

Function NEXT-NASSN gets a list of value assignments from BETAS and NGEN to be tested for consistency. The function returns a list of top-level value assignments.

Calls: add1base3, in, get-assn.

Called from: nested, validnest.

(buildbs ntable arws betas)

Function BUILDBS builds BETAS, the set of nested => operators occurring within the scope of a beta operator. The function has the side effect that NSTRUCT is updated to reflect nested => operators in the scope of alpha operators, and adds transitive arrows to ARROWS, a data structure indicating arrows which must co-occur, arrows which cannot co-occur, and transitive arrows. The function returns BETAS.

Calls: in, buildnested.

Called from: nested, buildnested.

(buildnested conds wj ntable arws betas)

Function BUILDNESTED is called in the case that nested => operators occur within the scope of alpha or beta operators. The function calls BUILDBS with appropriately updated NSTRUCT, ARWS, BETAS.

Calls: alphascope, buildbs, build, back, findrow, update-arws, wjconds, findlist, in.

Called from: buildbs.

(alphascope conds)

Function ALPHASCOPE checks for nested => operators within the scope of an alpha operator. The function returns T if such a => if found, and NIL otherwise.

Calls: apply-rules, init-nmust, in.

Called from: buildnested.

(init-nmust varvalues varval nmust)

Function INIT-NMUST initializes NMUST to variable assignments which must be in order that the given subformula be satisfiable. These elements are removed from NVALS. The function returns initialized NMUST and setq's NVALS.

Calls: varassn, front, back.

Called from: alphascope.

(wjconds wj conds wks)

Function WJCONDS returns a list of the worlds accessible from world wj given the list of conditions for accessibility from wj.

Calls: no other user-defined functions.

Called from: buildnested.

(update-arws arws wj w1conds wks)

Function UPDATE-ARWS adds transitive arrows to ARWS necessitated by nested => operators. The function returns updated ARWS.

Calls: trans.

Called from: updated-arws, buildnested.

(trans arws wj w1conds wk)

Function TRANS returns a list of the arrows required due to transitivity when nested => operators occur.

Calls: in, get-trans.

Called from: update-arws.

(get-trans arws wj wk awiwk)

Function GET-TRANS searches ARWS for arrows (A wi wj). The function returns a list of the arrows (A wi wk) required by transitivity.

Calls: modfarws.

Called from: trans.

(modfarws awiwj wk awiwk)

Function MODFARWS returns a list of transitive arrows (A wi wk) given a list of arrows (A wi wj).

Calls: no other user-defined functions.

Called from: get-trans.

(generate-test nstruct arrows)

Function GENERATE-TEST generates RTFC configurations and tests them for consistency. If a double inconsistency is found, the wff is valid, and the function returns NIL. Otherwise the configuration consisting of all worlds in one equivalence class is tested. If it is consistent, the false model is returned. Otherwise, if there is at least one not-arrow remaining, configurations along the not-arrow paths are tested. If a consistent not-arrow configuration is found, that false model is returned. Otherwise the remaining configurations are tested for consistency. If no consistent configuration is found, the wff is valid, and the function returns NIL. Otherwise the false model is returned. Thus function GENERATE-TEST returns NIL if the wff is valid, and a false model otherwise.

Calls: test-paths, rtnarws, test-arrows, remv-semiarws, init-necvals, get-worldlist, getfmodel, get-vars, patharws, init-paths, update-arrows.

Called from: validate, nested, validnest.

(test-paths nstruct necvals arrows paths)

Function TEST-PATHS tests configurations generated from each semi-complete structure of PATHS. The function returns a false model if the wff is invalid, and NIL otherwise.

Calls: consistentn, test-path, init-worldvals.

Called from: generate-test.

(init-worldvals nstruct path wvals)

Function INIT-WORLDVALS returns a list of the value assignments at each world which must be.

Calls: arrowp, findrow, update-wvals.

Called from: test-paths.

(update-wvals conds wj wvals nstruct arw)

Function UPDATE-WVALS updates WVALS to include the conditions at the given world WJ. It is determined whether WJ contains (1) a nested => within the scope if a beta, (2) a nested =>, or (3) no nested =>. Case (1) value assignments are made from BMODEL and nested-subformula value assignment rules applied, (2) value assignments are made from NSTRUCT and nested-subformula value assignment rules applied, and (3) value assignments are made from NSTRUCT.

Calls: incons-smust, in, nested-arrow, getfmodel, wigamma, findrow.

Called from: init-worldvals.

(nested-arrow arw nstruct)

Function NESTED-ARROW returns T if the given arrow (A wi wj) is a nested-arrow, that is, wj was created due to a => at wi, and NIL otherwise. This is to distinguish transitive-arrows from nested-arrows.

Calls: in, findrow.

Called from: update-wvals.

(wigamma condsatwj wivals)

Function WIGAMMA is applied in the case of nested-arrows (A wi wj), $i^{-}=1$, where wj was created due to the occurrence of a => at wi. Then that gamma-subformula at wi has the same value assignment at wj. The function returns the value assignment to that gamma.

Calls: no other user-defined functions.

Called from: update-wvals.

(test-path nstruct necvals arrows path worldvals)

Function TEST-PATH tests a given semi-complete structure for consistency. First the arrows and not-arrows on the path are tested for consistency. Then a configuration of added-arrows is generated and tested. Configuration generation and testing continues until a false configuration is found or all configurations have been tried. The function returns a false configuration if found, and NIL otherwise. If

WORLDVALS contains an inconsistent world, NIL is returned.

Calls: test-semicomp, gen&test-configs, get-wrlds.

Called from: test-paths.

(test-semicomp path nstruct)

Function TEST-SEMICOMP performs consistency tests on arrows and not-arrows of PATH, a semi-complete structure. Each not-arrow (wi (~ &)) is tested against (1) each other not-arrow (wi (~ &1)), (2) each arrow (A wi wj) separately, and (3) the conditions at wi (reflexivity test). Nested- and transitive-arrows (A wi wj), i ~= 1, and their return-arrows (A wj wi) are tested for consistency, and NECVALS and ARROWS updated accordingly. The function returns T if an inconsistency is found in one of the first three tests (the path is then inconsistent), and NIL otherwise.

Calls: incons-notarws, betanots, doublearw-incons, check-nested, checkarws.

Called from: test-path.

(betanots path)

Function BETANOTS updates ARROWS to reflect the occurrence of not-arrows with beta subformulas. The function returns NIL.

Calls: notarrowp, altassns.

Called from: test-semicomp.

(incons-notarws notarws path nstruct)

Function INCONS-NOTARWS performs not-arrow consistency testing, checking each not-arrow (wi (~ &)) against (1) each other not-arrow (wi (~&1)), (2) each arrow (A wi wj), and (3) the conditions at wi (reflexivity test). The function returns T if any inconsistency is found, and NIL otherwise.

Calls: notarrowp, incons-winots, get-winots, get-wiarws, remvwinots.

Called from: test-semicomp.

(notarrowp elem)

Function NOTARROWP returns T if the given ELEM is a not-arrow, and NIL otherwise.

Calls: no other user-defined functions.

Called from: betanots, incons-notarws, get-winots, remvwinots,incons-notarrows, get-notarws, case3beta, infix-notarws.

(get-winots wi path wis)

Function GET-WINOTS returns a list of ANDed not-arrows on PATH emanating from wi.

Calls: notarrowp, andwff.

Called from: incons-notarws, incons-notarrows.

(get-wiarws wi path wis)

Function GET-WIARWS returns a list of arrows (A wi wj) on PATH which emanate from wi.

Calls: arrowp.

Called from: incons-notarws, test-betas.

(remvwinots wi not-arws notarws)

Function REMVWINOTS returns NOTARWS from which not-arrows from wi have been removed.

Calls: notarrowp.

Called from: incons-notarws, incons-notarrows.

(incons-winots wi winotarws wiarws nstruct)

Function INCONS-WINOTS tests the mutual consistency of all not-arrows (wi (¯ A)) on PATH. Function INCONS-WINOT-ARWS is called to test the consistency of the not-arrows against each arrow (A wi wj) separately. Function INCONS-WINOTS returns T if an inconsistency is found, and NIL otherwise.

Calls: altassns.

Called from: incons-notarws.

(incons-winot-arws wi winotarws wiarws nstruct wvals)

Function INCONS-WINOT-ARWS tests the mutual consistency of not-arrows (wi (~ A)) and arrows (A wi wj), all from wi. WORLDVALS is updated to reflect the conditions which must be true in order for an arrow to be consistent with the not-arrows. The function returns T if the elements are mutually inconsistent, and NIL otherwise.

Calls: reflex-test, andwff, andform, transform, findrow, altassns, upd-worldvals.

Called from: incons-winots.

(upd-worldvals mustvals wj wvals)

Function UPD-WORLDVALS setq's WORLDVALS to reflect conditions required at wj due to a not-arrow (wi (~ &)) and arrow or nested-arrow (A wi wj). The function updates WORLDVALS and returns NIL.

Calls: findrow, front, back.

Called from: incons-winot-arws, reflex-test, reflexw1, nestedarw-incons.

(reflex-test wi winotarws condswi)

Function REFLEX-TEST tests the given NOTARW against the conditions at wi. If i=1 then, for the sake of efficiency, value assignments to NOTARW are made and appended to FALSEMODEL since that work has already been done for the conditions at world 1. WORLDVALS is updated to reflect the conditions which must be true at wi in order to be consistent with NOTARW. The function returns T if an inconsistency is found, and NIL otherwise.

Calls: reflexw1, andform, transform, altassns, andwff, upd-worldvals.

Called from: incons-winot-arws.

(reflexw1 notconds w1conds)

Function REFLEXW1 tests not-arrow (w1 (~ &)) against FALSEMODEL value assignments at world 1 for consistency. The function returns T if the two are inconsistent, and NIL otherwise.

Calls: altassns, andwff, upd-worldvals.

Called from: reflex-test.

(andform valassns subf)

Function ANDFORM returns a given set of value assigments translated to an ANDed formula.

Calls: andwff, transform.

Called from: incons-winot-arws, reflex-test, nestedarw-incons, test-rtnarw.

(transform valassn)

Function TRANSFORM returns a given VALASSN translated according to the following rules: (=> A C 1) := (=> A C); (A 1) := A; (=> A C 0) := (~ (=> A C)); (A 0) := (~ A).

Calls: no other user-defined functions.

Called from: incons-winot-arws, reflex-test, andform, nestedarw-incons, test-rtnarw.

(remv-nested conds)

Function REMV-NESTED returns the conditions for accessibility from a world wj with arrows to nested worlds removed.

Calls: in.

Called from: doublearw-incons, test-betas.

(doublearw-incons path nstruct nestednecs)

Function DOUBLEARW-INCONS performs consistency tests on nested-arrows on the given PATH. NESTED-NECS is updated with nested- and transitive-arrows and their return-arrows. The function returns T if the two are mutually inconsistent, and NIL otherwise.

Calls: arrowp, nestedarw-incons, remv-nested, findrow.

Called from: test-semicomp.

(nestedarw-incons wi wj condsfromwi condsfromwj nestednecs)

Function NESTEDARW-INCONS tests nested- and transitive-arrows (A wi wj), i~=1, for consistency. NESTED-NECS is updated accordingly. The function returns T if a double-arrow inconsistency is found, and NIL otherwise.

Calls: altassns, andwff, andform, transform, findrow, test-rtnarw, update-nested-necs, getfmodel, upd-worldvals.

Called from: doublearw-incons.

(test-rtnarw wvals nestednecs wi wj condsfromwj)

Function TEST-RTNARW tests return-arrow (A wj wi) for consistency. WORLDVALS and NESTED-NECS are updated to reflect the consistency/inconsistency of (A wj wi). The function returns T if an inconsistency arises which makes the whole path inconsistent, and NIL otherwise.

Calls: altassns, andwff, andform, transform, findrow, update-nested-necs, inconsistent, getfmodel.

Called from: nestedarw-incons.

(update-nested-necs arw mustvals nestednecs nestarw inconsist)

Function UPDATE-NESTED-NECS updates NESTED-NECS to reflect the consistency/ inconsistency of nested-arrows and their return-arrows and the necessary value assignments if consistent. The function returns updated NESTED-NECS.

Calls: in, findrow, front, back.

Called from: nestedarw-incons, test-rtnarw.

(checkarws nnecs narws)

Function CHECKARWS checks each arrow in NARWS for inconsistency or consistency dependent upon the co-occurrence of the return-arrow. NARWS is updated appropriately. The function returns updated NARWS.

Calls: in, arrowp.

Called from: test-semicomp.

(check-nested necvals path narws nnecs)

Function CHECK-NESTED compares NESTED-NECS and NECVALS for inconsistency in value assignments and appropriately updates NESTED-ARWS. The function returns T if an inconsistency is found which makes the entire path inconsistent, and NIL otherwise.

Calls: mustarws, in, arrowp, update-nestarws.

Called from: test-semicomp.

(mustarws path narws)

Function MUSTARWS setq's NESTED-ARWS to include nested and transitive arrows from PATH which must be present in any configuration from the current PATH. The function returns NIL.

Calls: no other user-defined functions.

Called from: check-nested.

(update-nestarws narws rowk nnecs necvals)

Function UPDATE-NESTARWS checks pairs of arrows (A wi wk) (A wj wk) for consistency of value assignments at wk. NESTED-ARWS is updated appropriately. The function returns updated NESTED-ARWS.

Calls: upd-nestarws, checkwks.

Called from: check-nested.

(checkwks rowk necrow narws)

Function CHECKWKS checks value assignments at the two given arrows (A wi wk) (A wj wk) for consistency. NARWS is updated appropriately. The function returns updated NARWS.

Calls: in, arrowp, cons-wks.

Called from: update-nestarws, upd-nestarws.

(cons-wks rowk necrow)

Function CONS-WKS returns F if value assignments at ROWK and NECROW are consistent, and NIL otherwise.

Calls: diffval.

Called from: checkwks.

(diffval valassn necrow)

Function DIFFVAL returns T if the given value assignments are inconsistent, and NIL otherwise.

Calls: no other user-defined functions.

Called from: cons-wks.

(upd-nestarws narws rowk necvals)

Function UPD-NESTARWS compares value assignments at ROWK of NESTED-NECS with those of NECVALS at wk and updates NESTED-ARWS appropriately. The function returns updated NESTED-ARWS.

Calls: checkwks.

Called from: update-nestarws.

(test-arrows necvals nstruct newnecvals)

Function TEST-ARROWS tests each arrow for consistency, and adds values which must be assigned to variables in order that the arrow be consistent to NECVALS. The function returns updated NECVALS.

Calls: front, back, find, update-necvals.

Called from: generate-test.

(update-necvals arw nstruct vals newnecvals)

Function UPDATE-NECVALS tests the given arrow ARW (A wi wj) for consistency. The AND of the conditions for worlds accessible from wi and the conditions at wj are tested. Any necessary values are added to VALS. The function returns updated VALS from NECVALS.

Calls: incons-smust, andwff, wiconds, findrow, check-wprime, add-vals, add-arw, next-sassn.

Called from: test-arrows.

(wiconds conds)

Function WICONDS returns the conditions for worlds accessible from wi with nested arrows removed, and (A wprime wi) arrows removed in the case of worlds with negative labels (worlds created due to a false gamma).

Calls: in.

Called from: update-necvals.

(check-wprime arw rtnarw vals newnecvals)

Function CHECK-WPRIME is called when arrow (A wi wj) has been found to be inconsistent. The function determines whether world wi was created from a false or true gamma. If from a true gamma, then VALS is returned with "incons" appended. If from a false gamma, then the arrow (A wj wi) is checked. If this arrow is inconsistent, VALS is returned with "incons" appended; if consistent, VALS is returned with "cons" and arrow (A wj wi) appended; if "checkrtnarw" appears in NECVALS for arrow (A wj wi), then "cons" and (A wj wi) are appended to NECVALS for arrow (A wi wj). If (A wj wi) has not yet been tested

for consistency, "checkrtnarw" is appended to NECVALS for arrow (A wi wj). The function returns updated VALS.

Calls: in, findrow.

Called from: update-necvals.

(add-vals smust vals)

Function ADD-VALS adds value assignments which must be the case in world wj in order that arrow (A wi wj) be consistent, to NECVALS. The function returns updated VALS.

Calls: in, add-val.

Called from: update-necvals.

(add-val val vals)

Function ADD-VAL adds value assignment VAL to the variable in VALS. The function returns updated VALS.

Calls: front, back, find.

Called from: add-vals.

(add-arw smust svals sprev smodel nullcount)

Function ADD-ARW checks all alternative value assignments for consistency. The function returns SPREV which is null if all alternatives are inconsistent, and equal to value assignments which must be the case if a consistent value assigment is found. The function differs from other functions designed to test alternatives in that it further determines whether there is exactly one consistent value assignment.

Calls: inconsistent, next-sassn, intersect.

Called from: update-wvals, update-necvals, test-notarws.

(intersect set1 set2)

Function INTERSECT calls function INTERSECTION with the shortest list as argument 1 in order to improve efficiency.

Calls: intersection.

Called from: add-arw.

(intersection set1 set2 isect)

Function INTERSECTION returns the intersection of lists SET1 and SET2.

Calls: no other user-defined functions.

Called from: intersect.

(remv-semiarws necvals newnecvals semiarws)

Function REMV-SEMIARWS returns NECVALS from which arrows of the semi-complete structure have been removed.

Calls: in, delet.

Called from: generate-test.

(patharws paths arws)

Function PATHARWS returns a list of the arrows in the semi-complete structures of PATHS.

Calls: add-arws.

Called from: generate-test.

(add-arws path arws)

Function ADD-ARWS adds arrows from PATH to ARWS omitting duplicates. The function returns updated ARWS.

Calls: in, arrowp.

Called from: patharws.

(arrowp elem)

Function ARROWP returns T if the given list is an arrow, and NIL otherwise.

Calls: no other user-defined functions.

Called from: init-worldvals, get-wiarws, doublearw-incons, checkarws, check-nested, add-arws, double-arw, remv-incons, incons->, check-necvals, beta-nestednecs, case2beta.

(rtnarws necvals newnecs)

Function RTNARWS checks NECVALS for arrows (A wi wj) containing "checkrtnarw." If the return arrow (A wj wi) is inconsistent, then (A wi wj) is inconsistent. Otherwise (A wi wj) is consistent. The function returns updated NECVALS.

Calls: in, awjwi.

Called from: generate-test.

(awjwi rtnarw arw newnecs)

Function AWJWI checks the return arrow (A wj wi) when a "checkrtnarw" occurs in NECVALS for arrow (A wi wj). If (A wj wi) is inconsistent, "checkrtnarw" is changed to "incons" for (A wi wj). Otherwise "checkrtnarw" is changed to "cons (A wj wi)." The function returns updated NEWNECS.

Calls: in, findrow, front, back.

Called from: rtnarws.

(update-arrows necvals arrows)

Function UPDATE-ARROWS checks NECVALS for inconsistent arrows, arrows which cannot co-occur, and arrows which must co-occur. The function returns updated ARROWS.

Calls: cant-co-occur, doube-arw.

Called from: generate-test.

(double-arw arrows ncv)

Function DOUBLE-ARW checks NECVALS for double-arrow inconsistencies. If such an inconsistency is found, ARROWS is updated, and any paths containing both worlds wi and wj are removed from PATHS. ARROWS is also updated to reflect necessary return-arrows. The function returns updated ARROWS.

Calls: in, findrow, updated-arrows, check-paths, arrowp, ifwiwj, ifwi.

Called from: update-arrows.

(ifwiwj wi wj arrows)

Function IFWIWJ updates ARROWS to reflect the occurrence of arrows (A -wi -wj) and (A -wj -wi) which must co-occur. The function returns updated ARROWS.

Calls: no other user-defined functions.

Called from: double-arw.

(ifwi wi wj arrows)

Function IFWI updates ARROWS to reflect the occurrence of arrows (A wi wj), (A wj wi) such that if (A wi wj) occurs, then (A wj wi) must occur. The function returns updated ARROWS.

Calls: no other user-defined functions.

Called from: double-arw.

(updated-arrows wi wj arrows paths)

Function UPDATED-ARROWS adds "imposbl" worlds wi, wj to ARROWS. Wi, wj are impossible in the sense that both cannot co-exist in any semi-complete structure. The function returns updated ARROWS.

Calls: no other user-defined functions.

Called from: double-arw.

(check-paths wi wj paths newpaths)

Function CHECK-PATHS returns PATHS with any paths containing impossible worlds wi and wj removed.

Calls: in.

Called from: double-arw.

(cant-co-occur arrows ncv)

Function CANT-CO-OCCUR updates ARROWS to reflect the occurrence of inconsistent arrows and arrows which cannot co-occur; e.g., (A wi wk) and (A wj wk) each require different value assignments to a variable, say A, at wk. The function returns updated ARROWS.

Calls: cannot, wkset, remvwkset.

Called from: update-arrows.

(wkset ncv worldk)

Function WKSET returns the set of arrows in NCV to world wk.

Calls: no other user-defined functions.

Called from: cant-co-occur.

(remvwkset wk ncv)

Function REMVWKSET returns NCV from which the set of arrows to world wk has been removed.

Calls: no other user-defined functions.

Called from: cant-co-occur.

(cannot worldk arrows)

Function CANNOT calls REMV-INCONS to remove inconsistent arrows from WORLDK and function CANT to check each pair of arrows for consistency. The function returns updated ARROWS.

Calls: remv-incons.

Called from: cant-co-occur.

(remv-incons set worldk arw)

Function REMV-INCONS returns the set of arrows to world wk from which inconsistent arrows have been removed. ARROWS is setq'd to indicate the occurrence of incon- sistent arrows.

Calls: in, arrowp.

Called from: cannot.

(cant worldk arrows)

Function CANT calls CHECK-PAIRS to check each pair of arrows in the set of consistent arrows to world wk for mutual consistency. If inconsistent, ARROWS is updated. The function returns updated ARROWS.

Calls: check-pairs.

Called from: cannot.

(check-pairs arw worldk arrows)

Function CHECK-PAIRS returns ARROWS updated to reflect the occurrence of pairs of arrows from a set of arrows to world wk which are mutually inconsistent.

Calls: diffvals, upd-arw.

Called from: cant.

(diffvals val1 val2)

Function DIFFVALS returns T if a variable has a different value in each given arrow, and NIL otherwise.

Calls: diffv.

Called from: check-pairs.

(diffv val1 val2)

Function DIFFV returns T if one of VAL1, VAL2 is assigned the value 1 and the other 0, and NIL otherwise.

Calls: no other user-defined functions.

Called from: diffvals.

(upd-arw arw1 arw2 arrows)

Function UPD-ARW updates ARROWS in the case that a pair of mutually inconsistent arrows is found. The arrows are checked to determine whether one or both of the world labels to world wk is negative. In this case return-arrows need to be examined, otherwise the arrows cannot co-occur. The function returns updated ARROWS.

Calls: no other user-defined functions.

Called from: check-pairs.

(init-paths nstruct)

Function INIT-PATHS generates the incomplete N-structure (before forward-connected arrows are added). The function returns data structure PATHS representing the incomplete diagram.

Calls: nested-gammas, get-conds, test-template, expanded, transitive, nested-paths.

Called from: generate-test.

(get-conds world conds newlist)

Function GET-CONDS returns a list of the worlds accessible from a world in which each world is a top-level element.

Calls: delet, find, checkdarw.

Called from: init-paths, nested-paths.

(checkdarw wi conds)

Function CHECKDARW returns listed CONDS if CONDS is of the form (v (A wi wj) (¯ a)) or (A wi wj), and NIL otherwise.

Calls: no other user-defined functions.

Called from: get-conds.

(test-template template newtemp nstruct)

Function TEST-TEMPLATE tests not-arrows for consistency. An inconsistent not- arrow is removed from TEMPLATE. The function returns updated TEMPLATE.

Calls: test-temp, get-ands, delet-ands.

Called from: init-paths.

(test-temp andlist orlist notas temp world nstruct)

Function TEST-TEMP checks each not-arrow within a given row of TEMPLATE for consistency with ANDLIST. The not-arrow is also checked with non-gamma conditions at the world from which it emanates. This is necessary because of the reflexive property. The function returns updated TEMP.

Calls: reflexive, elim-elem, elim¯path.

Called from: test-template.

(elim-elem e l)

Function ELIM-ELEM removes the given element E from the row in which it appears and returns the updated list L.

Calls: front, back, element-of, delet.

Called from: test-temp.

(reflexive notarw world nstruct)

Function REFLEXIVE tests not-arrows for consistency with the non-gamma conditions at the world from which it emanates. If inconsistent, the not-arrow will be removed from TEMPLATE. The function returns T if the not-arrow is inconsistent with the reflexive property, and NIL otherwise.

Calls: reflex1, altassns, andwff, findrow.

Called from: test-temp.

(elim~path notarw andlist nstruct)

Function ELIM~PATH tests a not-arrow for consistency with non-gamma conditions at worlds which must be accessible to the world from which the not-arrow emanates. The function returns T if the not-arrow is inconsistent, and NIL otherwise.

Calls: altassns, andwff, converted.

Called from: test-temp.

(andwff subf1 subf2)

Function ANDWFF returns the AND of the two given subformulas.

Calls: no other user-defined functions.

Called from: get-winots, incons-winot-arws, reflex-test, reflexw1, andform, nestedarw-incons, test-rtnarw, update-necvals, reflexive, elim~path, converted, test-notarws, test-betas, incons-betanots.

(converted arw semi nstruct)

Function CONVERTED converts an arrow to the conditions to be tested for consistency. An arrow (A wi wj) from the semi-complete structure is converted to the conditions which hold at wj. An added arrow (A wi wj) is converted to the AND of the conditions for accessibility from wi and the conditions at wj. The function returns the converted arrow.

Calls: findrow, andwff.

Called from: elim~path.

(reflex1 notarw)

Function REFLEX1 tests a not-arrow emanating from world 1 with the non-gamma conditions at world 1. The function returns T if the not-arrow is inconsistent, and NIL otherwise.

Calls: incons-smust, inconsistent, nsat, next-sassn.

Called from: reflexive.

(altassns subf)

Function ALTASSNS checks alternative assignments for consistency. The function returns T if the given SUBFormula is inconsistent, and NIL otherwise.

Calls: incons-smust, add-arw, next-sassn.

Called from: betanots, incons-winots, incons-winot-arws, reflex-test, reflexw1, nestedarw-incons, test-rtnarw, reflexive, elim~path.

(incons-smust subf)

Function INCONS-SMUST sets up the variables to generate value assignments for arrow-consistency testing. SVALS, SMUST, SGEN, and SPREV, are setq'd. The function returns T if SMUST is inconsistent, and NIL otherwise.

Calls: inconsistent, apply-rules, get-mustbe, remvar, init-gen.

Called from: update-wvals, update-necvals, reflex1, altassns, test-notarws, incons=>beta.

(nsat smust svals smodel)

Function NSAT tests each alternative value assignment for consistency. The function returns T if the value assignment is inconsistent, and NIL otherwise.

Calls: consistent, next-sassn.

Called from: reflex1, incons=>beta.

(next-sassn svals smust)

Function NEXT-SASSN gets a list of value assignments from SVALS and SGEN to be tested for consistency. The function returns a list of top-level value assignments.

Calls: add1base3, in, get-assn, next-sassn.

Called from: update-wvals, update-necvals, add-arw, reflex1, nsat, test-notarws, incons=>beta, b=>sat.

(transitive arws paths)

Function TRANSITIVE returns PATHS with transitive arrows added.

Calls: intrans.

Called from: init-paths.

(intrans arw paths newpaths)

Function INTRANS checks each row of paths for an arrow from arrows which requires the addition of a transitive arrow. The function returns PATHS with added transitive arrows.

Calls: in, front, back.

Called from: transitive.

(complete template paths)

Function COMPLETE returns PATHS with arrows added for nested =>'s and transitivity.

Calls: comp, expanded.

Called from: init-paths.

(comp wj exprow paths newpaths)

Function COMP returns PATHS expanded for added nested arrows and not-arrows.

Calls: inarw, newnrows, front, back.

Called from: complete.

(newnrows exprow path rows)

Function NEWNROWS returns a given row expanded due to the addition of nested arrows.

Calls: no other user-defined functions.

Called from: comp.

(inarw wj plist)

Function INARW returns T if an arrow (A wi WJ) occurs in PLIST, and NIL otherwise.

Calls: no other user-defined functions.

Called from: comp.

(nested-paths nstruct template)

Function NESTED-PATHS returns TEMPLATE updated to reflect the occurrence of nested => operators.

Calls: in, get-conds.

Called from: init-paths.

(get-worldlist nstruct worlds)

Function GET-WORLDLIST returns a list of the worlds in NSTRUCT.

Calls: no other user-defined functions.

Called from: generate-test.

(get-vars nwff vars)

Function GET-VARS returns a list of sublists consisting of the propositional variables in the given wff.

Calls: propvar, op.

Called from: generate-test, test-config.

(op nwff)

Function OP returns T if the given subformula is an operator, and NIL otherwise.

Calls: in.

Called from: get-vars.

(init-necvals wj worlds vars necvals)

Function INIT-NECVALS initializes data structure NECVALS, a list of all possible arrows and values of variables which must be in order for an arrow to be consistent. The function returns NECVALS initialized to a table consisting of a column of arrows, and columns for each propositional variable in the formula.

Calls: wjs.

Called from: generate-test.

(wjs wj worlds vars wjlist)

Function WJS returns a list of the arrows to world wj and the propositional variables in the given wff.

Calls: no other user-defined functions.

Called from: init-necvals.

(expanded olist)

Function EXPANDED returns PATHS initialized so that each row is a path with no OR's.

Calls: orpath, newrows.

Called from: init-paths, complete.

(orpath plist)

Function ORPATH returns T if the given list contains an OR path, and NIL otherwise.

Calls: no other user-defined functions.

Called from: expanded.

(newrows row)

Function NEWROWS is given a path from TEMPLATE and returns new rows derived from the path in which each row is a path containing no OR's.

Calls: allpaths, get-ands, delet-ands, get-genlist.

Called from: expanded.

(get-ands row andlist)

Function GET-ANDS returns a list of nodes on the path which must be, exclusive of any OR's.

Calls: no other user-defined functions.

Called from: test-template, newrows.

(delet-ands row newr)

Function DELET-ANDS returns a row of NOTARWS with AND's removed and only OR's remaining.

Calls: no other user-defined functions.

Called from: test-template, newrows.

(get-genlist olist glist)

Function GET-GENLIST returns a list of 0's of length equal to the length of the list of OR's.

Calls: no other user-defined functions.

Called from: newrows.

(allpaths world andlist orlist genlist newlist)

Function ALLPATHS returns a list of all paths containing no OR's derived from the given row of paths.

Calls: add1base2, get-path.

Called from: newrows.

(get-path world orlist genlist path)

Function GET-PATH returns a path with no OR's given a list of OR's and AND's.

Calls: no other user-defined functions.

Called from: allpaths.

(in e l)

Function IN determines whether E is an element of list L at any level of nesting of sublists. The function returns T if E is in L, and NIL otherwise.

Calls: in-any.

Called from: incons-gamma, incons, propvar, propwff, next-assn, nested-gammas, remvdups, remvdup, get-worlds, count, next-nassn, buildbs, buildnested, build, alphascope, trans, update-wvals, nested-arrow, remv-nested, update-nestednecs, checkarws, check-nested, wiconds, check-wprime, add-vals, remv-semiarws, add-arws, rtnarws, awjwi, double-arw, check-paths, remv-incons, next-sassn, intrans, nested-paths, op, incons+, incons~, incons->, check-necvals, findworlds, get-varassns, remv-mustvals, verif=> 1, verify-flsconfig, verify-true=>, verify-false=>, orfalse=>, applies, incons-betanecs, incons-beta, incons=>beta, infix-assns.

(in-any e l)

Function IN-ANY finds each sublist at each level of nesting to determine whether or not E is in L. The function returns a concatenated list of T's. The number of T's in the returned list is equal to the number of times E occurs in L. If E is not in L, then NIL is returned.

Calls: no other user-defined functions.

Called from: in.

(front e l)

Function FRONT returns the head of a list L up to, but not including, element E of the list L.

Calls: no other user-defined functions.

Called from: incons-gamma, init-mustbe, modifiedt, modifiedf, count, init-nmust, upd-worldvals, upd-alterns, update-nested-necs, test-arrows, add-val, awjwi, elim-elem, intrans, comp, delet, del, next-config, next-world, check-necvals, upd-wrldvals, gen&test-configs, update-altern.

(back e l)

Function BACK returns the tail of a list L beginning with the element succeeding element E to the last element of list L.

Calls: no other user-defined functions.

Called from: init-mustbe, modifiedt, modifiedf, count, buildnested, init-nmust, upd-worldvals, update-nested-necs, test-arrows, add-val, awjwi, elim-elem, intrans, comp, delet, del, upd-wrldvals, upd-alterns.

(find elem l)

Function FIND searches L for any substructure whose CAR is equal to ELEM. The function returns the first such substructure, or NIL if no such substructure is found.

Calls: findlist.

Called from: count, test-arrows, add-val, get-conds, valsubf.

(findn num elem l)

Function FINDN searches L for any substructure whose CAR is equal to ELEM. The function returns the nth such substructure, assuming an index base of 1, or NIL if no such substructure is found.

Calls: findlist.

Called from: no function of program VALIDATE.

(findlist elem l)

Function FINDLIST searches L for any substructure whose CAR is equal to ELEM. The function returns a concatenated list of all occurrences of such substructures, or NIL if no such substructure is found.

Calls: no other user-defined functions.

Called from: incons, buildnested, find, findn.

(findrow elem l)

Function FINDROW searches L for a substructure whose CAR is equal to ELEM. The function returns the first such substructure, or NIL if no such substructure is found.

Calls: no other user-defined functions.

Called from: gamma-true, gamma-false, buildnested, init-worldvals, update-wvals, nested-arrow, incons-winot-arws, upd-worldvals, doublearw-incons, nestedarw-incons, test-rtnarw, update-nested-necs, update-necvals, check-wprime, awjwi, double-arw, reflexive, converted, incons-necvals, check-necvals, upd-wrldvals, test-notarrows, verif=>1, verif=>0, update-altern, verify-true=>, verify-false=>, no-access, accesswj, test-betas, incons-betanots, incons-beta, incons=>beta.

(remvlist elems l)

Function REMVLIST returns a copy of list L from which each element of ELEMS has been removed.

Calls: no other user-defined functions.

Called from: inconsistent, gen&test-configs, next-config, b=>sat.

(delet s l)

Function DELET returns a copy of list L from which element S has been deleted. S need not be a top-level element of L. If S occurs more than once in L, then the first occurrence of S is deleted from L.

Calls: del, front, back.

Called from: remv-semiarws, get-conds, elim-elem.

(del s l newl)

Function DEL is called from DELET in the case where S is not a top-level element of L. The function returns the new list NEWL, which is the list L from which S has been deleted.

Calls: element-of, front, back.

Called from: delet.

(element-of s l)

Function ELEMENT-OF returns the substructure of L in which S is a top-level element.

Calls: no other user-defined functions.

Called from: elim-elem, del, access-worlds.

(gen&test-configs path necvals arrows worldvals worlds config)

Function GEN&TEST-CONFIGS generates and tests configurations for consistency. It is determined whether added-arrows are required. If so, sets of arrows are generated which, in combination with PATH, form RTFC configurations. Each configuration is tested for consistency. The cycle is repeated until a consistent configuration is found, or all configurations have been tested and found to be inconsistent. The function returns a false configuration, if found, and NIL otherwise.

Calls: test-config, get-arwset, elemn, next-config, remvlist, flat, front.

Called from: test-path.

(next-eqtemp eqtemp eqs ws level)

Function NEXT-EQTEMP determines the next equivalence-class template from the previous one. The function returns the new EQUIVTEMP.

Calls: init-eqtemp, elemn, sumtop, top.

Called from: next-config.

(elemn n l)

Function ELEMN returns the nth element of the given list L.

Calls: no other user-defined functions.

Called from: gen&test-configs, next-eqtemp, next-config, next-world.

(init-eqtemp c w equiv)

Function INIT-EQTEMP returns an equivalence-class template of C classes for W worlds.

Calls: no other user-defined functions.

Called from: next-eqtemp.

(top l len)

Function TOP returns the first LEN elements in list L.

Calls: init-class.

Called from: next-eqtemp.

(sumtop l lsum)

Function SUMTOP returns the sum of the top LEN elements in L.

Calls: no other user-defined functions.

Called from: next-eqtemp.

(init-class remworlds leng)

Function INIT-CLASS returns a list of the top leng elements in the given list REMWORLDS.

Calls: no other user-defined functions.

Called from: top, init-config, next-world.

(next-config config class size worlds remworlds)

Function NEXT-CONFIG returns the next combination of worlds in equivalence classes, given the previous combination. If all combinations of worlds have been tried for the current equivalence-class template, the next equivalence-class template is determined, and a combination returned.

Calls: next-world, elemn, init-config, next-eqtemp, remvlist, flat, front.

Called from: gen&test-configs.

(flat l)

Function FLAT returns a list of the atoms in the given list L.

Calls: no other user-defined functions.

Called from: gen&test-configs, next-config, access-worlds.

(init-config eqtemp worlds newconfig)

Function INIT-CONFIG setq's CONFIG to the initial configuration given a new equivalence-class template and a set of worlds.

Calls: init-class.

Called from: next-config.

(next-world eqclass size pos worlds)

Function NEXT-WORLD determines the next worlds to be included within an equivalence class. The function returns NIL if all combinations of worlds within the class have been tried, and the new class otherwise.

Calls: elemn, front, init-class.

Called from: next-config.

(c1 wrlds worlds arwset)

Function C1 returns a list of arrows generated from an equivalence class, i.e., the worlds within an equivalence class are fully-connected. The function returns NIL if the equivalence class consists of 0 or 1 worlds).

Calls: update-arwset.

Called from: get-arwset.

(update-arwset w worlds arwset)

Function UPDATE-ARWSET adds arrows from W to worlds in WORLDS to ARWSET.

Calls: no other user-defined functions.

Called from: c1.

(get-wrlds path worlds)

Function GET-WRLDS returns a list of the worlds (other than world 1) on PATH.

Calls: no other user-defined functions.

Called from: test-path.

(get-arwset config arwset)

Function GET-ARWSET returns the set of arrows generated from the given CONFIG.

Calls: no other user-defined functions.

Called from: gen&test-configs.

(ci class1 classi arwset)

Function CI returns the set of arrows from each world in CLASS1 to each world in the other i classes.

Calls: upd-arwset.

Called from: get-arwset.

(upd-arwset class1 classi arwset)

Function UPD-ARWSET returns the set of arrows from each world in CLASS1 to each world in CLASSI.

Calls: get-arws.

Called from: ci.

(get-arws w classi arwset)

Function GET-ARWS returns the set of arrows from given world W to each world in CLASSI.

Calls: no other user-defined functions.

Called from: upd-arwset.

(test-config arwset config path necvals arrows worldvals worlds)

Function TEST-CONFIG tests the consistency of the given configuration generated from CONFIG and PATH. The constraints placed on arrows (stored in ARROWS) in any consistent configuration are compared with ARWSET to determine whether any constraints are violated. Necessary values from NECVALS are added to WRLDVALS, a copy of WORLDVALS updated with value assignments which must be under the current configuration. WORLDVALS contains values which must be under the current PATH. Not-arrows (wk (~ &)) are tested against the conditions at world wi in the case that k ~= 1 and (A wk wi) occurs in ARWSET but not in PATH. If the configuration is consistent, then VERIFY-FALSECONFIG is called to verify the gamma and beta subformulas. The function returns a false configuration if found, and NIL otherwise.

Calls: incons-arwset, incons-necvals, incons-notarrows, verify-falseconfig, get-betas, beta-nestednecs, get-alternatives, getfmodel, get-vars.

Called from: gen&test-configs.

(incons-arwset arwset arrows worlds)

Function INCONS-ARWSET checks ARWSET for consistency with the conditions in ARROWS. The function returns T if an inconsistency is found, and NIL otherwise.

Calls: incons~, incons->, incons+.

Called from: test-config.

(incons+ mustarw arwset)

Function INCONS+ returns T if MUSTARW is not in ARWSET, and NIL otherwise.

Calls: in.

Called from: incons-arwset.

(incons~ imposbl arwset)

Function INCONS~ returns T if ARWSET contains the impossible world(s) in IMPOSBL, and NIL otherwise. If IMPOSBL contains more than one world, it means that these worlds cannot co-occur.

Calls: in.

Called from: incons-arwset.

(incons-> condl arwset worlds)

Function INCONS-> returns T if ARWSET contains the antecedent of CONDL but not the consequent, and NIL otherwise.

Calls: in, arrowp.

Called from: incons-arwset.

(incons-necvals arwset necvals wvals)

Function INCONS-NECVALS adds value assignments from NECVALS to WRLDVALS for each arrow in ARWSET. The function returns T if an inconsistency is found, and NIL otherwise.

Calls: check-necvals, findrow.

Called from: test-config.

(check-necvals w necrow wvals)

Function CHECK-NECVALS checks for consistency the value assignments of NECROW and the value assignments of WRLDVALS. If consistent, the value assignments of NECROW are added to WRLDVALS. The function returns T if an inconsistency is found, and NIL otherwise.

Calls: in, arrowp, inconsistent, front, findrow, upd-wrldvals, remvnull.

Called from: incons-necvals.

(upd-wrldvals mustvals wj wvals)

Function UPD-WRLDVALS setq's WRLDVALS to reflect conditions required at wj due to a not-arrow (wi (~ &)) and arrow or nested-arrow (A wi wj). The function updates WRLDVALS and returns NIL.

Calls: findrow, front, back.

Called from: check-necvals, test-notarws.

(incons-notarrows arwset path wrldvals notarws worlds)

Function INCONS-NOTARROWS tests for consistency not-arrows (wk (~ &)) against the conditions at world wi in the case that k ~= 1 and (A wk wi) occurs in ARWSET but not in PATH. The function returns T if inconsistent, and NIL otherwise.

Calls: notarrowp, test-notarrows, get-winots, findworlds, remvwinots.

Called from: test-config.

(findworlds wk worlds path arwset wrlds)

Function FINDWORLDS finds worlds wi for which arrow (A wk wi) occurs in ARWSET but not in PATH. The function returns a list of such worlds, or NIL if none occur.

Calls: in.

Called from: incons-notarrows.

(test-notarrows notconds wk wrldvals wrlds)

Function TEST-NOTARROWS tests not-arrows (wk (~ &)) and the conditions at world wi in the case that k ~= 1 and (A wk wi) appears in ARWSET but not in PATH. In other cases the (~ &) and conditions at wi would have been tested previously. The function returns T if an inconsistency is found, and NIL otherwise.

Calls: test-notarws, findrow.

Called from: incons-notarrows.

(test-notarws notconds condswi wi wivals wk)

Function TEST-NOTARWS tests for consistency not-arrows from WK and the conditions at world WI where (A wk wi) is in ARWSET but not in PATH. If consistent, WRLDVALS is updated. The function returns T if an inconsistency is found, and NIL otherwise.

Calls: incons-smust, andwff, inconsistent, upd-wrldvals, add-arw, next-sassn.

Called from: test-notarrows.

(get-betas arws betarws)

Function GET-BETAS returns ARWS adjusted so that only beta occurrences remain.

Calls: no other user-defined functions.

Called from: test-config.

(beta-nestednecs path arws)

Function BETA-NESTEDNECS returns ARROWS with beta markers to nested-arrows from PATH and their return-arrows added.

Calls: arrowp.

Called from: test-config.

(remvnull necassns assns)

Function REMVNULL returns a list of value assignments from which null assignments have been removed.

Calls: no other user-defined functions.

Called from: check-necvals.

(get-alternatives wrldvals vars alterns)

Function GET-ALTERNATIVES returns a list of alternative value assignments to propositional variables at each world in the configuration. Variable assignments which must be from WRLDVALS are appended to the alternatives for the remaining unassigned variables.

Calls: get-alterns, get-varassns.

Called from: test-config.

(get-alterns mustvals vars)

Function GET-ALTERNS gets a list of alternative assignments to variables at a world. The variable assignments which must be (from WRLDVALS) are appended to the alternatives for the remaining unassigned variables. The function returns a list of alternative variable assignments.

Calls: get-alts, remv-mustvals, get-valist.

Called from: get-alternatives.

(get-alts mustvals vars alterns vals)

Function GET-ALTS returns a list of alternative value assignments to variables at a world. The variable assignments which must be (from WORLDVALS) are appended to the alternatives for the remaining unassigned variables.

Calls: update-alterns, add1base2.

Called from: get-alterns.

(update-alterns vars vals mustvals alterns altern)

Function UPDATE-ALTERNS returns ALTERNS with an alternative value assignment to variables appended.

Calls: no other user-defined functions.

Called from: get-alts.

(get-valist varlength valist)

Function GET-VALIST returns a list of 0's of length VARLENGTH.

Calls: no other user-defined functions.

Called from: get-alterns.

(get-varassns wivals varassns)

Function GET-VARASSNS returns a list of value assignments to variables as distinct from assignments to gamma subformulas. The function returns NIL if WIVALS contains no variable assignments at world wi.

Calls: in.

Called from: get-alternatives.

(remv-mustvals mustvals vars)

Function REMV-MUSTVALS returns a list of VARS from which variables in MUSTVALS ;have been removed.

Calls: in.

Called from: get-alterns.

(subval subf)

Function SUBVAL takes a SUBFormula in which values have been substituted for propositional variables and returns the value assignment for the SUBFormula.

Calls: subf-rules, determ.

Called from: incons-betaconds.

(subf-rules opr opd1 opd2)

Function SUBF-RULES takes an operator and one or two operand values, and returns a value for the subformula.

Calls: no other user-defined functions.

Called from: subval.

(determ opr opd1 opd2)

Function DETERM returns T if a value for the subformula can be determined with knowledge of the value of only one operand, and NIL otherwise. For example, if either operand is false, (^ A B) is false.

Calls: no other user-defined functions.

Called from: subval.

(valsubf subf varassns)

Function VALSUBF returns SUBFormula in which all occurrences of propositional variables have been replaced with their respective values from VARASSNS.

Calls: propvar, find.

Called from: incons-betaconds.

(get-notarws wi path wis)

Function GET-NOTARWS returns a list of not-arrows from world wi.

Calls: notarrowp.

Called from: no function of program VALIDATE.

(build-accessbl wvals path arwset accessbl)

Function BUILD-ACCESSBL determines the gammas in WVALS for which accessible worlds wj exist making the value assigned gamma at wi hold. The function returns a list of such accessible worlds for each world wi in WVALS.

Calls: get-accessbl.

Called from: verify-falseconfig.

(get-accessbl wi wivals path arwset accessbl)

Function GET-ACCESSBL determines the gammas in WVALS for which accessible worlds wj exist making the value assigned the gamma at wi hold. The function returns a list of such accessible worlds.

Calls: verif=>1, verif=>0.

Called from: build-accessbl.

(verif=>1 wi gamma* arwlist path)

Function VERIF=>1 determines whether a world wj exists which makes a gamma at wi true. In this case there appears: 1) arrow (A wi wj) in ARWSET, 2) arrow (A wi wj) in PATH if i=1, 3) wi=wj, or 4) not-arrow (wi (~ &)). For gamma (=> &1 &2 1), the conditions at world wj are (^ &1 &2) or there exists a not-arrow (wi (~ &1)). The function returns T if such a world wj or not-arrow (wi (~ &1)) is found, and NIL otherwise.

Calls: findrow, in, verify=>1.

Called from: get-accessbl.

(verify=>1 wi gamma* arwlist)

Function VERIFY=>1 determines whether a world wj exists which makes the GAMMA* true at world WI. In this case the arrow (A wi wj) appears in ARWLIST. For gamma (=> &1 &2 1), the conditions at wj are (^ &1 &2). The function returns T if wj exists, and NIL otherwise.

Calls: no other user-defined functions.

Called from: verif=>1.

(verif=>0 wi gamma* arwlist)

Function VERIF=>0 determines whether a world wj exists which makes a gamma at wi false. In this case arrow (A wi wj) appears in ARWSET, or if i=1, then the arrow appears in PATH, or it may be the case that i=j. For gamma (=> &1 &2 0), the conditions at wj are (^ &1 (~ &2)). The function returns T if such a world wj is found, and NIL otherwise.

Calls: findrow, verify=>0.

Called from: get-accessbl.

(verify=>0 wi gamma* arwlist)

Function VERIFY=>0 determines whether a world wj exists which makes GAMMA* false at world WI. In this case the arrow (A wi wj) appears in PATH and for gamma (=> &1 &2 0), the conditions at wj are (^ &1 (~ &2)). The function returns T if wj exists, and NIL otherwise.

Calls: no other user-defined functions.

Called from: verif=>0.

(remv-access accessbl wvals newvals)

Function REMV-ACCESS returns WVALS from which gammas have been removed if there exists an accessible world wj, and the conditions at wj make the value assigned the gamma at wi hold.

Calls: update-newvals.

Called from: verify-falseconfig.

(update-newvals accessbl wivals)

Function UPDATE-NEWVALS returns WIVALS from which gammas in ACCESSBL have been removed.

Calls: no other user-defined functions.

Called from: remv-access.

(next-altern altern vals)

Function NEXT-ALTERN returns an alternative list of value assignments to propositional variables at each world in WRLDVALS.

Calls: no other user-defined functions.

Called from: verify-fconfig.

(update-altern altern alterns)

Function UPDATE-ALTERN modifies ALTERN so that the next call to NEXT-ALTERN gets a new ALTERNative value assignment to propositional variables at worlds in WRLDVALS. The function returns updated ALTERN.

Calls: front, findrow.

Called from: verify-fconfig.

(verify-falseconfig arwset config path arrows wrldvals alterns)

Function VERIFY-FALSECONFIG verifies that the given configuration ARWSET is indeed a false configuration. The function has two goals: 1) to verify that the values of gamma subformulas actually hold by checking for accessible worlds (or reflexive accessibility), and 2) to verify that the value assignments meet with conditions imposed by beta subformulas which may not appear in WRLDVALS. The function returns the false model if found, and NIL otherwise.

Calls: verify-flsconfig, build-accessbl, remv-access.

Called from: test-config.

(verify-flsconfig arswet config path arrows accessbl wvals alterns)

Function VERIFY-FLSCONFIG determines whether further verification is required. If a beta marker appears in ARROWS or an unverified => in WVALS, then function VERIFY-FCONFIG is called to do the verification. Otherwise a false configuration is returned.

Calls: in, get-fconfig, merge, verify-fconfig.

Called from: verify-falseconfig.

(verify-fconfig arwset config path arrows accessbl wvals altern alterns)

Function VERIFY-FCONFIG verifies that the given configuration ARWSET is indeed a false configuration. VERIFY-FCONFIG is called in the case of a configuration composed of more than two worlds. The function has two goals: 1) to verify that the values of gamma subformulas actually hold by checking for accessible worlds (or reflexive accessibility), and 2) to verify that the value assignments meet with conditions imposed by beta subformulas which may not appear in WRLDVALS. The function returns the false model if found, and NIL otherwise.

Calls: verify, get-fconfig, merge, next-altern, update-altern.

Called from: verify-flsconfig.

(verify arwset config path arrows accessbl wvals altern)

Function VERIFY attempts to verify that the given false configuration is indeed false. Since WRLDVALS contains only value assignments which must be at each world, there may be constraints incurred by beta subformulas which do not appear in WRLDVALS. For example, given (^ (v A B) C 1) only (C 1) appears in WRLDVALS, but (v A B 1) disallows ((A 0)(B 0)). If a gamma subformula is nested within a beta, the gamma must also be verified. The function returns T if the ALTERNative value assignments at each world are consistent with the betas and gammas, and NIL otherwise.

Calls: verify-wvals=>, test-betas.

Called from: verify-fconfig.

(verify-wvals=> wvals altern accessbl arwset config)

Function VERIFY-WVALS=> returns T if for each world in WVALS, the values assigned gammas are verified, and NIL otherwise.

Calls: verify-wval=>.

Called from: verify.

(verify-wval=> wi gams altern accessbl arwset config)

Function VERIFY-WVAL=> returns T if for each gamma at world wi, the value assigned the gamma is verified, and NIL otherwise.

Calls: verify=>.

Called from: verify-wvals=>, b=>sat.

(get-fconfig model arwset path)

Function GET-FCONFIG formats the configuration for printing.

Calls: inorder, infix-assns, infix-notarws, getfmodel.

Called from: verify-flsconfig, verify-fconfig.

(infix-notarws nots infix)

Function INFIX-NOTARWS converts not-arrows from prefix to infix notation so that the output is in a more readable form. The function returns a list of arrows and not-arrows in which the not-arrow subformulas are in infix form.

Calls: notarrowp, inorder.

Called from: get-fconfig.

(infix-assns assns infix)

Function INFIX-ASSNS converts gamma-subformula in a false model from prefix to infix notation so that the output is in a more readable form. The function returns the false model with gamma-subformulas in infix form.

Calls: in, infix-gammas.

Called from: get-fconfig.

(infix-gammas assns infix)

Function INFIX-GAMMAS returns a list of variable assignments and gamma-subformula assignments in which gamma-subformulas have been converted from prefix to infix form.

Calls: inorder.

Called from: infix-assns.

(merge altern wvals accessbl newlist)

Function MERGE merges lists ALTERN, WVALS, AND ACCESSBL so that value assignments from each list for a world are combined and duplicates removed. The function returns a list containing value assignments from each given list for each world.

Calls: getfmodel.

Called from: verify-flsconfig, verify-fconfig.

(verify=> wi gam altern accessbl arwset config)

Function VERIFY=> attempts to verify the value assigned a gamma subformula at world wi. The function returns T if the value assigned the gamma is consistent with the value assignments to variables at other worlds in the configuration, and NIL otherwise.

Calls: verify-true=>, verify-false=>.

Called from: verify-wval=>.

(verify-true=> wi gam altern accessbl config)

Function VERIFY-TRUE=> attempts to verify that the given gamma subformula GAM is indeed true. There are three cases in which the gamma (=> &1 &2) is true at wi: 1) ((&1 1)(&2 1)) is consistent at wi, and there exists no accessible wj, (A wi wj), where ((&1 1)(&2 0)) holds, 2) (& 0) holds at all accessible worlds, i.e., there exists no wj, (A wi wj), where (& 1) holds, or 3) there exists accessible world wj, (A wi wj), where ((&1 1)(&2 1)) and there exists no accessible world wk, (A wj wk), where ((&1 1)(&2 0)). The function returns T if either of the three cases occur, and NIL otherwise.

Calls: in, findrow, consistent, no-access, access-worlds, foundwj.

Called from: verify=>.

(verify-false=> wi gam altern accessbl config)

Function VERIFY-FALSE=> attempts to verify that the given gamma subformula GAM is indeed false. There are two cases in which the gamma (=> &1 &2) is false at world wi: 1) ((&1 1)(&2 0)) holds at world wi and there exists no world wk, (A wi wk), where ((&1 1)(&2 1)), or if such a world wk does exist, then (A wk wi) is also in the configuration, or 2) there exists an accessible world wj, (A wi wj), where ((&1 1)(&2 0)) holds and there exists no world wk, (A wj wk), where ((&1 1)(&2 1)) holds, or if there exists such a world wk, then (A wk wj) is also present in the configuration. The function returns T if the gamma is verified, and NIL otherwise.

Calls: in, findrow, consistent, accesswj, access-worlds, orfalse=>, or=>false.

Called from: verify=>.

(access-worlds wi config)

Function ACCESS-WORLDS returns a list of worlds accessible to world wi, i.e., a list of worlds wj such that (A wi wj) occurs in the configuration.

Calls: element-of, flat.

Called from: verify-true=>, verify-false=>, foundwj, no-accesswk, or=>false.

(no-access conds accworlds altern)

Function NO-ACCESS attempts to verify that there exists no world wj, (A wi wj), where CONDS hold. The value assignments to variables at each accessible world wj is tested for consistency with the forbidden CONDS. The function returns T if CONDS is consistent with the variable assignments at no accessible world wj, and NIL otherwise.

Calls: consistent, findrow.

Called from: verify-true=>, no-accesswk.

(accesswj conds accworlds altern wjs)

Function ACCESSWJ returns a list of worlds wj such that (A wi wj), and CONDS is consistent with ALTERN at wj. The function returns NIL if no such world wj is found.

Calls: consistent, findrow.

Called from: verify-false=>, foundwj, or=>false.

(foundwj wi conds config altern)

Function FOUNDWJ attempts to verify that a gamma subformula (=> &1 &2) is true in the case that there exists a world wj accessible to wi, (A wi wj), where condition ((&1 1)(&2 1)) holds, and there exists no world wk accessible to wj, (A wj wk), where ((&1 1)(&2 0)) holds. The function returns T if the gamma subformula is found to be true, and NIL otherwise.

Calls: accesswj, access-worlds, no-accesswk.

Called from: verify-true=>.

(no-accesswk conds wjs accworlds altern config)

Function NO-ACCESSWK determines whether there exists some world wj in WJS such that there exists no world wk accessible to wj, (A wj wk), where CONDS is consistent with ALTERN at wk. The function returns T if such a world is found, and NIL otherwise.

Calls: no-access, access-worlds.

Called from: foundwj.

(orfalse=> wj wks arwset)

Function ORFALSE=> returns T if for each world wk in WKS accessible to wj, (A wk wj) occurs in ARWSET, and NIL otherwise.

Calls: in.

Called from: verify-false=>, or=>false.

(or=>false wjs wks conds arwset config altern)

Function OR=>FALSE determines whether, for some world wj in WJS, either there exists no accessible world wk in WKS where ((&1 1)(&2 1)) holds, or if there is such a world wk, then (A wk wj) is also in ARWSET. The function returns T if such a world wj is found, and NIL otherwise.

Calls: orfalse=>, accesswj, access-worlds.

Called from: verify-false=>.

(test-betas arwset config path arrows accessbl wvals altern)

Function TEST-BETAS attempts to verify that the given ALTERNative set of value assignments to variables is consistent with beta constraints on variable value assignments which would not appear in WRLDVALS. For example, given (^ (v A B) C 1) only (C 1) appears in WRLDVALS, but (v A B 1) disallows ((A 0)(B 0)). There are four cases in which betas may occur: 1) in the conditions at a world wi, 2) in an arrow in NECVALS, 3) in not-arrows (wk (~&)) where (A wk wi) occurs in PATH, and 4) in not-arrows (wk (~ &)) where (A wk wi) occurs in ARWSET but not in PATH. Case (4) is already taken care of by case (1) if the beta lies in wi and/or case (3) if the not-arrow contains a beta. If a gamma is nested within a beta, then function VERIFY=> is called to verify the value assigned the gamma. The function returns T if the value assignments of ALTERN are consistent with betas in ARROWS, and NIL otherwise.

Calls: case1beta, applies, findrow, incons-beta, case2beta, incons-betanecs, andwff, remv-nested, case3beta, incons-betanots.

Called from: verify.

(applies case type arwlist)

Function APPLIES determines whether the given CASE of a beta applies in the current configuration. For example, CASE1 beta applies only if the world is in PATH, CASE2 if the arrow is in ARWSET, CASE3 if the not-arrow is in PATH, and CASE4 would already have been checked in CASE1 and/or CASE3. The function returns T if the beta applies, and NIL otherwise.

Calls: in.

Called from: test-betas.

(case1beta marker)

Function CASE1BETA returns T if the given beta marker is of type CASE1, i.e., a beta in the conditions at a world, and NIL otherwise.

Calls: no other user-defined functions.

Called from: test-betas.

(case2beta marker)

Function CASE2BETA returns T if the given beta marker is of type CASE2, i.e., a beta in an arrow from NECVALS, and NIL otherwise.

Calls: arrowp.

Called from: test-betas.

(case3beta marker)

Function CASE3BETA returns T if the given beta marker is of type CASE3, i.e., a beta in a not-arrow on PATH, and NIL otherwise.

Calls: notarrowp.

Called from: test-betas.

(incons-betanots wk conds arwset config path accessbl wvals altern)

Function INCONS-BETANOTS verifies betas in not-arrows from WK at each world wi for which (A wk wi) occurs in ARWSET. INCONS=>BETA is called if a => occurs in the not-arrow, and INCONS-BETACONDS is called otherwise. The function returns T if for some world wi, the ALTERNative value assignment and beta in the not-arrow are inconsistent, and NIL otherwise.

Calls: incons-beta, andwff, findrow.

Called from: test-betas.

(remv-wiarws wi arws arwset)

Function REMV-WIARWS removes arrows emanating from world WI in ARWS. The function returns the remaining set of arrows.

Calls: no other user-defined functions.

Called from: incons-betanots.

(incons-betanecs wj wi conds arwset config path accessbl wvals altern)

Function INCONS-BETANECS verifies betas in arrows (A wj wi) in NECVALS. If j is less than 0 and the arrow inconsistent with ALTERN, then the beta is still consistent if (A wi wj) also occurs in ARWSET. The function returns T if the beta is inconsistent with the given ALTERNative value assignmnent to variables, and NIL otherwise.

Calls: incons-beta, in.

Called from: test-betas.

(incons-beta wi conds arwset config path accessbl wvals altern)

Function INCONS-BETA determines whether a => occurs in CONDS. If so, function INCONS=>BETA is called which returns T if the ALTERNative value assignment is inconsistent with the beta, and NIL otherwise. If no => occurs, function INCONS-BETACONDS is called which also returns T if the ALTERNative value assignment is inconsistent with the beta, and NIL otherwise.

Calls: in, incons=>beta, incons-betaconds, findrow, upd-alterns.

Called from: test-betas, incons-betanots, incons-betanecs.

(incons-betaconds conds altrow)

Function INCONS-BETACONDS verifies that a formula containing a beta subformula is consistent with the ALTERNative value assignment to variables in the case that no =>'s occur in CONDS. The function returns T if CONDS and ALTROW are inconsistent, and NIL otherwise.

Calls: subval, valsubf.

Called from: incons-beta.

(incons=>beta wi conds arwset config path accessbl wvals altern)

Function INCONS=>BETA verifies that a formula containing a beta subformula is consistent with the ALTERNative value assignment to variables in the case that a => occurs in CONDS. The function returns T if CONDS and ALTERN, WVALS, and ACCESSBL are inconsistent, and NIL otherwise.

Calls: incons-smust, inconsistent, findrow, upd-alterns, in, b=>sat, next-sassn, nsat.

Called from: incons-beta.

(b=>sat smust svals smodel wi altern accessbl arwset config)

Function B=>SAT attempts to verify that a set of value assignments is consistent in the case that a gamma subformula occurs within a beta. The function returns T if an inconsistency is found, and NIL otherwise.

Calls: inconsistent, next-sassn, remvlist, getfmodel, verify-wval=>.

Called from: icons=>beta.

(upd-alterns alternrow alternsrow)

Function UPD-ALTERNS removes from ALTERNS a set of value assignments to variables at row wi of ALTERNS when such a row is found to be inconsistent with beta constraints at wi. The function SETQ's ALTERNS and returns T.

Calls: front, back.

Called from: incons-beta, incons=>beta.

References

Beth, E. W., *The Foundations of Mathematics*, North-Holland, 1959, revised edition 1964.

Brachman, R. J., "'I Lied about the Trees' or Defaults and Definitions in Knowledge Representation," *The AI Magazine*, vol 6, no. 3, 1985, pp 80-93.

Burgess, J.P., "Quick Completeness Proofs for Some Logics of Conditionals," *Notre Dame Journal of Philosophy, vol. 22, no. 1, 1981, pp 76-84.*

Chellas, B.F., "Basic Conditional Logic," *Journal of Philosophical Logic* 4, 1975, pp 133-153.

Church, A., *Introduction to Mathematical Logic I*, Princeton, 1956.

Cottrell, G., "Re: Inheritance Hierarchies with Exceptions," *AAAI Workshop on Nonmonotonic Reasoning*, New Patz, New York, October 1984, pp 33-56.

Delgrande, J., "A Foundational Approach to Conjecture and Knowledge," Ph.d. thesis, Dept. of Computer Science, University of Toronto, Ontario, 1985.

Delgrande, J., "A Propositional Logic for Natural Kinds," AI-*86, Sixth Canadian Conference on Artificial Intelligence*, Montreal, Canadian Society for Computational Studies of Intelligence, May 1986, pp 44-48.

Delgrande, J., "A First-Order Conditional Logic for Prototypical Properties," to appear, *AI Journal*, 1987.

de Sousa, R., "The Natural Shiftiness of Natural Kinds," *Canadian Journal of Philosophy, vol. XIV, no. 4, 1984, pp 561-580.*

Etherington, D., and Reiter, R., "On Inheritance Hierarchies with Exceptions," *Proceedings AAAI-83*, 1983, pp 104-108.

Fahlman, S., *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, 1979.

Fahlman, S., Touretsky, D., and van Roggen, W., "Cancellation in a Parallel Semantic Network," *Proceedings of the Seventh International Conference on AI, Vancouver, B.C., 1981, pp 257-263.*

Fitting, M.C., *Proof Methods for Modal and Intuitionistic Logics*, Reidel Publishing, Dordrecht, Holland, 1983.

Gentzen, G., "Investigations into Logical Deduction," in *The Collected Papers of Gerhard Gentzen*, M.E. Szabo (ed.), North-Holland Publishing Company, Amsterdam, 1969, pp 68-131.

Hadley, R., "Model-Theoretic vs. Procedural Semantics," Technical Report LCCR TR 87-10, School of Computing Science, Simon Fraser University, 1987.

Hardegree, G., "The Conditional in Quantum Logic," in P. Suppes (ed.), *Logic and Probability in Quantum Mechanics*, Reidel Publishing, Dordrecht, Holland, 1976.

Hintikka, J., "Form and Content in Quantification Theory," *Acta Philosophica Fennica*, vol. 8, 1955, pp 7-55.

Holte, R., and Wharton R., "Generative Structure in Enumerative Learning Systems," AI-*86, Sixth Canadian Conference on Artificial Intelligence*, Montreal, Canadian Society for Computational Studies of Intelligence, May 1986, pp 11-16.

Hughes, G.E., and Cresswell, M.J., *An Introduction to Modal Logic*, Methuen and Col. Ltd., 1968.

Jennings, R., "The Natural Conditional," presented at the *Western Canadian Philosophical Association*, October 1983.

Jennings, R., "An Eddy in the Theory of Conditionals," in preparation.

Kleene, S.C., *Introduction to Mathematics*, Princeton: Van Nostrand, 1952.

Kripke, S., "Semantical Considerations on Modal Logics," *Acta Philosophica Fennica, Modal and Many-valued Logics*, 1963, pp 83-94.

Lesperance, Y., "Handling Exceptional Conditions in PSN," *Proceedings of the Third Conference of the Canadian Society for Computational Studies of Intelligence*, Victoria, B.C., 1980, pp 63-70.

Lewis, D., *Counterfactuals*, Harvard University Press, 1973.

McCarthy, J., "Circumscription - A Form of Non-Monotonic Reasoning," *Artificial Intelligence, vol. 13, pp 27-39, 1980.*

McDermott, D., and Doyle, J., "Non-Monotonic Logic I," *Artificial Intelligence*, vol. 13, 1980, pp 41-72.

Mittelstaedt, P., *Quantum Logic*, Reidel Publishing, Dordrecht, Holland, 1978.

Moore, R.C., "Semantical Considerations on Nonmonotonic Logic," *Proceedings IJCAI-83*, Karlsruhe, 1983, pp 272-279.

Minsky, M., "A Framework for Representing Knowledge," in *The Psychology of Computer Vision*, P.H. Winston (ed.), McGraw-Hill, 1975, pp 211-277.

Nute, D., "Counterfactuals," *Notre Dame Journal of Formal Logic, vol. 16, 1975, 476-482.*

Nute, D., *Topics in Conditional Logic*, Philosophical Studies Series in Philosophy, vol. 20, Reidel Publishing, Dordrecht, Holland, 1980.

Oppacher, F., and Suen, E., "An Efficient Tableau-based Theorem Prover," AI-*86, Sixth Canadian Conference on Artificial Intelligence*, Montreal, Canadian Society for Computational Studies of Intelligence, May 1986, pp 31-35.

Pollock, J., *Subjunctive Reasoning*, Philosophical Studies Series in Philosophy, Reidel Publishing, Dordrecht, Holland, 1976.

Putnam, H., "Is Semantics Possible?," in *Naming, Necessity, and Natural Kinds*, S.P. Schwartz (ed.), Cornell University Press, 1977, pp 102-118.

Quine, W.V., "Natural Kinds," in *Naming, Necessity, and Natural Kinds*, S.P. Schwartz (ed.), Cornell University Press, 1977, pp 155-175.

Reiter, R., "A Logic for Default Reasoning," *Artificial Intelligence* 13, 1980, pp 81-132.

Reiter, R., "On Reasoning by Default," *Proceedings of the Second Symposium on Theoretical Issues in Natural Language Processing*, Urbana, Illinois, July 25-27, 1978.

Reiter, R., and Criscuolo, G., "On Interacting Defaults," *Proceedings IJCAI-81*, Vancouver, B.C., 1981, pp 270-276.

Reiter, R., and Criscuolo, G., "Some Representational Issues in Default Reasoning," *Computation and Mathematics with Applications*, 9(1), 1983, pp 15-27.

Rosch, E., "Principles of Categorization," in *Cognition and Categorization*, E. Rosch and B.B. Lloyds (eds.), Lawrence Erlbaum Associates, 1978.

Salmon, N., *Reference and Essence*, Princeton, N.J., Princeton University Press, 1981.

Schwartz, S., "Natural Kind Terms," *Cognition*, vol. 7, 1979, pp 301-315.

Smullyan, R.M., "A Unifying Principle in Quantification Theory, *Proceedings of the National Academy of Sciences*, June 1963.

Smullyan, R.M., *First-Order Logic*, Springer-Verlag, Berlin, 1968.

Stalnaker, R.F., "A Theory of Conditionals," in *Studies in Logical Theory*, N. Rescher (ed.) Basil Blackwell, Oxford, 1968, pp 98-112.

Touretsky, D., "Inheritable Relations: A Logical Extension to Inheritance Hierarchies," *Theoretical Approaches to Natural Language Understanding*, 1984.

van Benthem, J., "Foundations of Conditional Logic," *Journal of Philosophical Logic, vol. 13, 1984, pp 303-349.*

van Fraassen, B.C., "The Logic of Conditional Obligation," *Journal of Philosophical Logic I*, 1972, pp 417-438, 1972.

Veltman, F., *Logics for Conditionals*, Ph.d. thesis, University of Amsterdam, Netherlands, 1985.