修 士 論 文

非単調推論を用いた分散診断システム

指導教官　東条敏 助教授

北陸先端科学技術大学院大学
情報科学研究科情報処理学専攻

510052 小藤 義行

1997 年 2 月 14 日

要 旨

近年, ネットワーク技術の発達等により, 各種システムは大規模化, 分散化傾向にある. それに伴い, システムのある場所に異常が生じた際に発生するアラームは増大し, それらを人手で処理するには非常に熟練した技術や膨大な時間を要する. そこで, これら問題を解決するために故障診断システム等の研究がなされている. 本研究は, そのうちマルチエージェント環境における分散診断システムモデル [1] に基づいた, 頑健性のある診断システムの構築を目的とする. また, 故障原因の推論に非単調推論を用い, システムの頑健性の向上に貢献する事を示す.

目 次

図 目 次

第 1 章

はじめに

1.1　目的・背景

深層知識の利用

診断型システムは, 1970 年代にアメリカで集中的に研究された医療診断システムにはじまり, 1980 年代からは故障診断システムとしても研究されてきた. 診断型システムにおける診断とは, 対象とするシステムが, 正常な場合に期待されるものとは異なる動作をした場合に, 観測されたデータやシステムに関する知識等を用いてその原因を同定することを指す. この原因同定に使われる知識に, 経験的に得られる “表層知識” と, システムの設計情報に基づく “深層知識” がある. これらは各々長所と短所を合わせ持つ. “表層知識” の利用は効率的な推論を可能にしてくれるが, 知識ベースを作成することが難しく, また全ての故障パターンを網羅できるものではないため, 未知の事態のが生じたときにはシステムが無力化する恐れがある. “深層知識” の利用は診断の完全性を保証するが, システムの動作パターンを全てテストするため推論の負荷が問題となる. 近年では完全性を保ちつつ診断の効率化をはかるため, 深層知識の利用をベースにして, その推論過程の省略のために表層知識を用いるシステムが提案されている. しかし前述の通り表層知識には理論的根拠がなく, 未知の状況に対してはシステムが無力化する恐れがある. したがって, 深層知識を用いて, その推論の負荷を分散させるシステムが求められている.

診断対象の大規模化・分散化

ネットワークオペレーションやプラントといった大規模で分散している対象を診断する場合, 人手で処理を行うには診断対象を熟知しているなど熟練した技術が必要である. また熟練した技術を持っていたとしても, 診断対象が分散環境にある場合, 物理的な距離の隔たりや機能分散に起因するデバックの難しさ等の理由から, 異常原因同定には非常に多くの時間を消費する. それらの理由から, 故障診断システム等の研究がなされている. 診断システムの研究の中に, 診断対象を構成要素に分け, 各構成要素毎に 1 つの診断エージェントが監視し, 異常が生じた時には診断エージェント間の協調によって原因を同定する枠組 [1] がある. この枠組は, システム全体の負荷分散, 情報欠落に対する診断システムの頑健性, システムの柔軟性に優れていて, 本枠組の基となるものである. 本枠組ではそのうちの頑健性に注目し, 情報欠落だけでなく誤情報に対する頑健性の実現を試みる. また, 不確実な知識の利用に起因する推論過程の非単調性に備えるために非単調推論を導入する. そして, 提案したモデルを並列オブジェクト指向言語 ABCL/f を用いて実装し, 1 プロセッサ 1 エージェントのマルチエージェント環境でモデルの機能の実証を行う.

1.2 論文の構成

本稿はこの章を含め 5 つの章からなる. 次の 2 章では, 分散システムの概説とその故障形態, 耐故障性について述べる. 3 章では分散システムの頑健性と拡張性を念頭においた分散診断システムを提案し, 4 章でシステムの実装環境を説明し, 各種の実験を通して頑健性, 並列処理による効率化等のの実証を行う. そして 5 章でまとめをおこなう.

第 2 章

マルチエージェントとシステムの頑健性

本章では, 本研究の目的である分散システムの頑健性の背景知識を述べる. 章構成としては, 前半で分散システムの利点とマルチエージェントシステムの紹介を行った後, 本研究の基となった「診断エージェントを持つエージェントの協調による分散診断」[1] を紹介し, 後半で分散システムの故障形態の紹介と本研究の目的である誤情報に対する頑健性の実現に必要なデフォルト推論 [14] と相互認識ネットワークモデル [23] の説明を行う.

2.1 分散システム

近年, コンピュータ処理の大規模化に伴い, 従来の集中型のシステムでは, 負荷集中による処理効率の低下, システムの頑健性の低下等の弊害が生じている. そのため, 従来の集中型システムから, システム全体の負荷を分散させ耐故障性の向上につながる分散型システムへの移行が進められている.

分散型システムの利点をまとめると以下のようになる.

負荷分散: 分散システムでは, 負荷がシステム全体に分散されることから, より複雑で大量な仕事が可能となる.

冗長性: 分散システムでは, 複数な場所で同じ処理が可能なため, システムの部分的な故障に対して頑健性を持つ.

並列処理: 分散システムでは, 処理を並列に行う事が出来る. そのため, 処理の時間短縮が期待できる.

分散要素の独立性: 分散システムでは構成要素の独立性が高くなる. そのため, システムの透過性が増し, 故障場所の特定や, システムの拡張性が高くなる.

分散型システムを実現するための理論の研究の一つに分散人工知能とマルチエージェントがある. 上で挙げた利点は, これら二つの枠組に共通するものである. 次節ではマルチエージェントと分散人工知能の違いについて述べ, システムの頑健性に注目した場合, どちらが適しているかについて考察する.

2.2 分散問題解決とマルチエージェント

分散人工知能とマルチエージェントの明確な区別はなされていない. 本稿では, 文献 [7] の区別を基本としている.

2.2.1 分散問題解決

分散問題解決は, 解決すべき問題を適切に分割し並列処理することで問題解決の効率化をはかることを目的とする. 従って, 与えられた問題を適切に分割し各分散要素に割り当てるプロセスと, 各分散要素の処理結果をまとめて決断を下すプロセスが必要となる. 分散問題解決は, 以下に示す 2 種類のものに分類される [20].

1. タスク共有

与えられた問題が幾つかの区域に分割されるとき, その部分区域の問題をエージェントが分担して解くことで負荷分散をはかる枠組をタスク共用の分散問題解決と呼ぶ. このとき各エージェントは担当区域の部分問題解決を目指す. タスク共用の具体例として DSS がある.

DSS(Distributed Sensing System)

DSS は, 監視区域内の乗物地図を作ることを目的としている. 従ってその仕事は, 乗物の認識, クラス分け, 追跡等である. これら仕事をこなすために 3 種類のエージェントが定義されている.

モニタ: 他の全てのエージェント情報を統括するエージェント. 問題分割も担当する.

プロセッサ: センサー情報の解析を行うエージェント. 全体の監視区域中の一部分区域の情報解析を担当する. (より多くの情報を得るために, 適切な範囲を監視するセンサーと契約する機能を持つ.)

センサー: 低レベルの信号解析を行うエージェント.

DSS において共有されるタスクは, 担当区域の監視である. 監視の効率化のために, センサーとそのセンサー情報解析を部分区域に分散させている. (図 2.1)

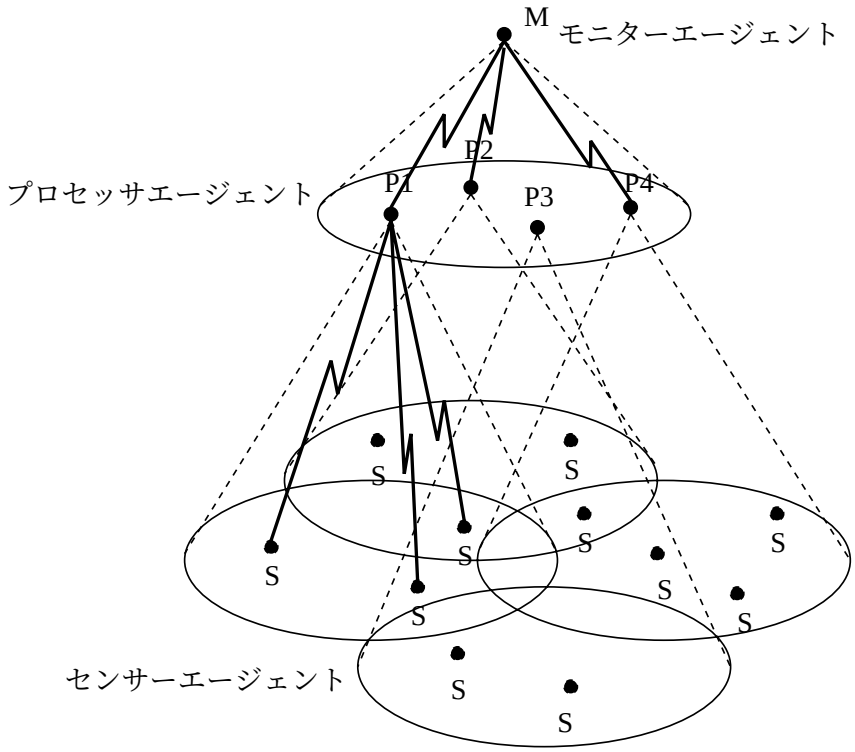

図 2.1: DSS のイメージ (文献 [12] より転載)

契約ネット

タスク共有の枠組において, タスクを分割し, 適当なエージェントに割り振るプロトコルに契約ネットがある [20].

エージェント定義: 契約ネットにおけるタスク提供側をマネージャ, 応答側を契約者と呼ぶ. 契約ネットのおおまかな流れは, マネージャから契約者に対しタスク提示がされ,

入札メッセージが契約者からマネージャへ, そしてマネージャから契約者に落札メッセージが送信されて契約完了となる.

相互選択: 契約ネットでは, 契約者はマネージャを, またマネージャは契約者を各々入札, 落札の時点で選択することになる. このしくみは相互選択と呼ばれ, 後に説明するマルチエージェントに繋がる要素である.

契約ネットプロトコルは先の例の DSS にも用いられている. 例えば, プロセッサエージェントは各種処理に有利なセンサーエージェントと “契約” することになる. 図 2.1 において契約は太線で表現されている. 従って, 処理によって契約するエージェントが異なることもあり得る. 逆にセンサーの立場から見れば, 自身の能力を一番発揮出来るプロセッサエージェントと契約することになる. (相互選択)

2. 結果共有

同じ目的をもつ複数のエージェントが協調して問題解決にあたるとき, その協調形態は結果共有と呼ばれる. 各エージェントは, 他のエージェントの情報と自身が持つ情報を用いて担当する部分問題の解決を目指す. 代表的な例に Hearsay-II[21] で採用されている黒板モデルがある.

黒板モデル

Hearsay-II は音声認識システムで, 単語数は 1000 語に限定されているが, 連続して発話される文を 90 %の正確さで認識可能である.

一般に音声認識の過程では, 音声波形から音節, 音節の列から単語, さらに単語から文を構成していく. しかし, 入力される音声波形に曖昧性があるため, 1 発話に対して処理の各段階で複数の仮説が生成される. これら各段階の仮説生成は, 知識源とよばれるエージェントによってなされ, 知識源間の黒板を通した協調によって仮説が搾られていく. 黒板モデルは以下の要素から成り立つ.

黒板: 黒板の定義は, 問題解決過程で生じる仮説を格納する知識源共通のメモリである. 各知識源は黒板の情報をもとに仮説を生成していく. Hearsay-II における黒板は, 入力波形, 音節, 単語, 文等の 7 レベルに分けられた階層構造をしている. 各レベルには, 様々な解釈が仮説として格納され, 各仮説には信頼度を示す数値が付加される.

知識源: 問題解決に必要な知識は知識源とよばれる独立したエージェントに各々格納されている. 知識源は音声認識過程の各段階に存在し, その起動条件は黒板上に成立する仮説である. 知識源は発火されると新たな仮説を生成したり, 存在する仮説を変更したりする. 中には, 下位レベルの仮説を統合して上位の仮説を生成するものや, 仮説間の制約条件に合わない仮説を削除するなどの働きをするものがある.

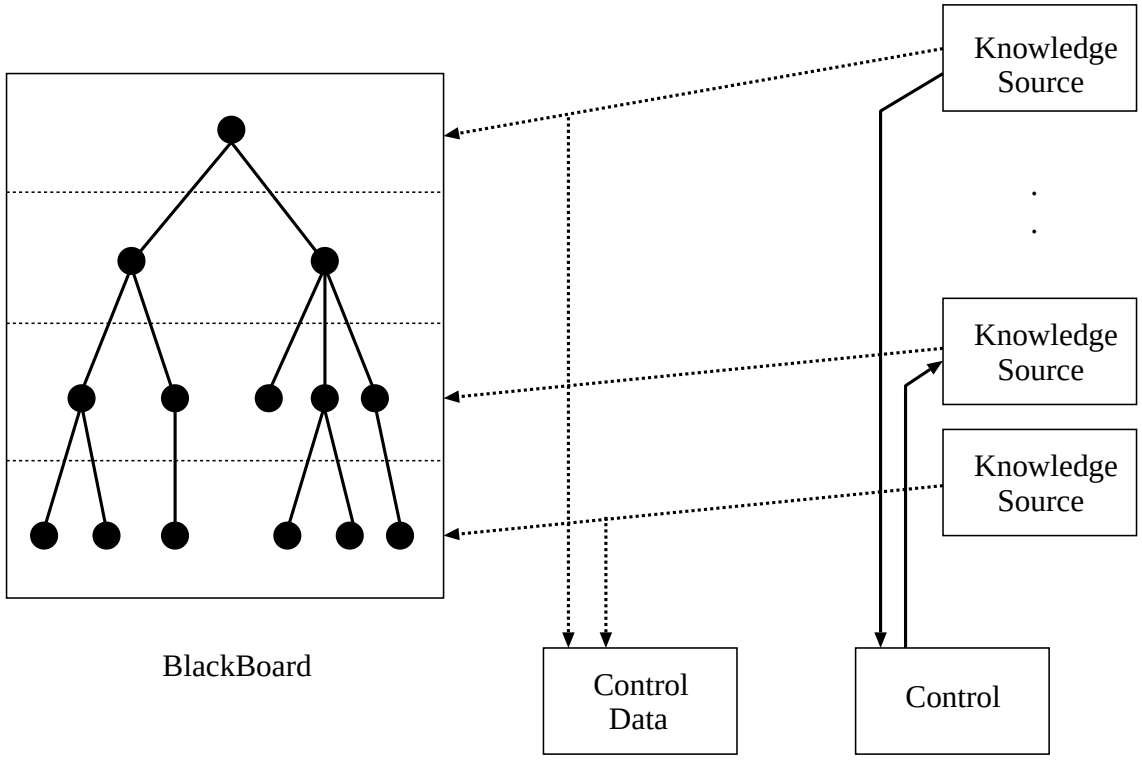

図 2.2: 黒板モデルのイメージ (文献 [12] より転載)

2.2.2 マルチエージェント

システムの分散要素であるエージェントが「自己」を持つ場合, そのシステムは分散人工知能ではなくマルチエージェントと呼ぶべきである. 共通の目的を持つ分散要素群が自己を持ち, 自身の利益を考え始めるとそれら分散要素は各自異なる目標を目指す. このとき, 自分に関する情報を他のエージェントに与えることは必ずしも自分の利益ならないの

で, マルチエージェントにおける各エージェントは常に, 他のエージェントの部分的な情報をもとに自己の利益を最大にすることを考えて行動することになる.

従ってマルチエージェントにおける協調とは, 自己の利益追求が基本であり, 場合によってはある程度の自己利益を犠牲にする事でシステム全体の利益を増加させ, 結果的には自己利益に還元されることを目指すものである. つまりマルチエージェントにおける協調の基本命題は, 「複数のエージェントが相互に協調活動を行うことで, 協調しない時よりもより高い利益が得られるか」となる.

ここで, 先の契約ネットについて分散人工知能とマルチエージェントの違いを考えてみる. 契約ネットでは, タスク要求を受けて実際に処理する契約者は入札時に, またタスク要求するマネージャは落札時に「自身に有利」な相手を選ぶことになる. (相互選択) この考え方はそれ自身で既にマルチエージェントと呼ぶにふさわしいかもしれない. しかし, マネージャと契約者を分けた考えは真のマルチエージェントとは言い難い. この形式の場合ではマネージャの価値があまりにも大きすぎるのである. 真に “マルチ” であるとはエージェント間格差が殆んど無い事を示している. エージェント間の格差を無くすと, 各エージェントはマネージャでありかつ契約者であることが自然となる. 従って, マルチエージェント環境における契約ネットプロトコルでは, 各エージェントは問題に応じてマネージャ, 契約者を演じ分ける事になる.

2.2.3 まとめ

マルチエージェントの導入

分散問題解決とマルチエージェントの違いは, 分散要素が自己を持つか持たないかである. 自己を持ち自己の利益を最大にする努力は, 各エージェントの自己防御に繋がり, 結果としてシステム全体の耐故障性の向上に貢献する. また, 機能の役割分担が明確な分散人工知能においては, 1 エージェントの故障がある処理プロセスの故障を意味するのに対し, 機能ではなく担当範囲を分担し, 各エージェントの機能については殆んど差がないマルチエージェントでは, 1 エージェントの故障の影響は, ある範囲の情報が欠落するだけに留まる.

システムの頑健性を比べる例として, 各枠組のエージェントが故障した場合を考える. 分散人工知能で推論や判断を行うエージェントが故障したとする. このとき, 他のエージェントは全て別の機能を果たしているため, あらかじめシステムに冗長性を持たしておくなど

の対策がなければシステムは機能しない. 一方, マルチエージェントにおいてエージェントが故障したらどうなるだろうか. マルチエージェントでは同じ機能を持ったエージェントが複数存在する. このため 1 エージェントの故障が推論機能を始めとする各種機能を奪う事はない. つまり, その影響は故障エージェントが担当する範囲の情報が欠落するに留まるのである.

従って, システムのより高い信頼性を目指すためには分散人工知能よりマルチエージェントが適しているとの判断は妥当であろう. 本研究ではこの判断に習いマルチエージェントシステムを導入する.

マルチエージェント環境における分散診断システム

先の分類において分散診断システムは結果共有の枠組になる. 結果共有では黒板モデル等が考えられているが, その各エージェントは知識源と呼ばれる “専門家” である. 従って, システム外部からの情報を各知識源がリレー方式に処理していくことになる. リレー方式では 1 エージェントの故障脱落により, システム自体が機能しなくなる可能性を持つ. この問題は, 黒板モデルにおける処理のリレー方式をやめ, エージェントが各々の担当範囲で一連の処理を行うタスク共有型の枠組を導入することで解決される. 各エージェントは, 担当範囲の局所診断を行い, 担当範囲外の情報が欲しいときはその担当エージェントとメッセージ通信する事で対処することになる. 更に, エージェントに自己を持たせた場合には, メッセージ通信時に相手エージェントの情報が自己の利益に繋がるとき以外は無視するなどの判断がなされることになる.

従って, マルチエージェント環境による分散診断システムとは, 結果共有かつタスク共有な分散問題解決だと定義できる.

2.3 診断モデルを持つエージェントの協調による分散診断

マルチエージェント環境における分散診断システムの研究に「診断モデルを持つエージェントの協調による分散診断」[1] がある. この枠組は, 診断対象の各構成要素を 1 エージェントが担当し局所診断を行うマルチエージェント環境をモデル化したものである. また, 故障診断に深層知識を用いている. 本研究はこの枠組に基づいている.

2.3.1 診断モデル

各診断エージェントは, 原因推論のために診断モデルを持っている. 診断モデルの定義は以下のようになされている. 診断対象の設計知識を用いて事象間の因果関係を導出し, 原因 → 結果 (図 2.3 左) 形式の知識を作成し, これを基に根には異常原因 (A), その異常が引き起こす中間現象 (B, C), そして最終的に現れる異常兆候 (D, E, F, G, H) という構造をもつ原因木を作成する. (図 2.3 右)

原因木は全て AND 木で定義されている. 一般に原因木では, AND 関係だけでなく OR 関係や EXOR 関係も考えられるが, OR は木を分割すればよく, また EXOR は AND と NOT で表す事が可能である.

診断モデルの葉はセンサーからの数値情報そのものではなく, センサー情報を知識ベースで変換したものである. たとえば, 温度センサー情報から入力データが時間とともに増加している場合, それらの情報は知識ベースによって「温度上昇」と置き換えられる. 診断モデルの葉にはこのようにして得られる現象が置かれている.

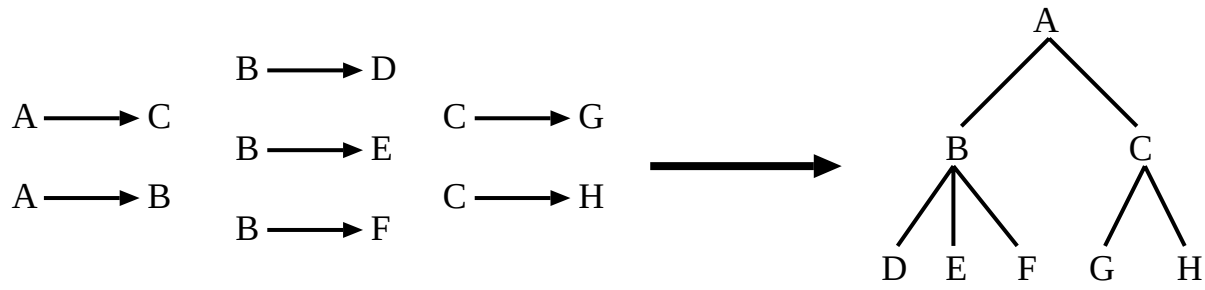

図 2.3: 診断モデル (文献 [1] より転載)

2.3.2 システムの概要

システムの概要は以下のようになる.

- 診断対象は, プラント, 分散 OS, 通信システム, 交信システム等.
- 各構成要素毎に, 1 エージェントが監視する.

- 診断エージェントは, センサー情報を解析する知識と, 異常原因を推論する診断モデルを持つ. 診断モデルは, 根が原因, node が中間現象, 葉が異常兆候を表す AND 木である.
- センサー入力は数値データ. 入力情報はセンサー入力だけでなく, 診断の材料になるものならなんでもよい.
- 各診断エージェントは互いに通信が可能.
- 各診断エージェントは複数の診断モデルを保持している.
- 推論は事象駆動型の仮説推論 [2] として行われる. 結果として出力されるのは, 原因に対してどれだけの割合の異常兆候が検出されたかである.

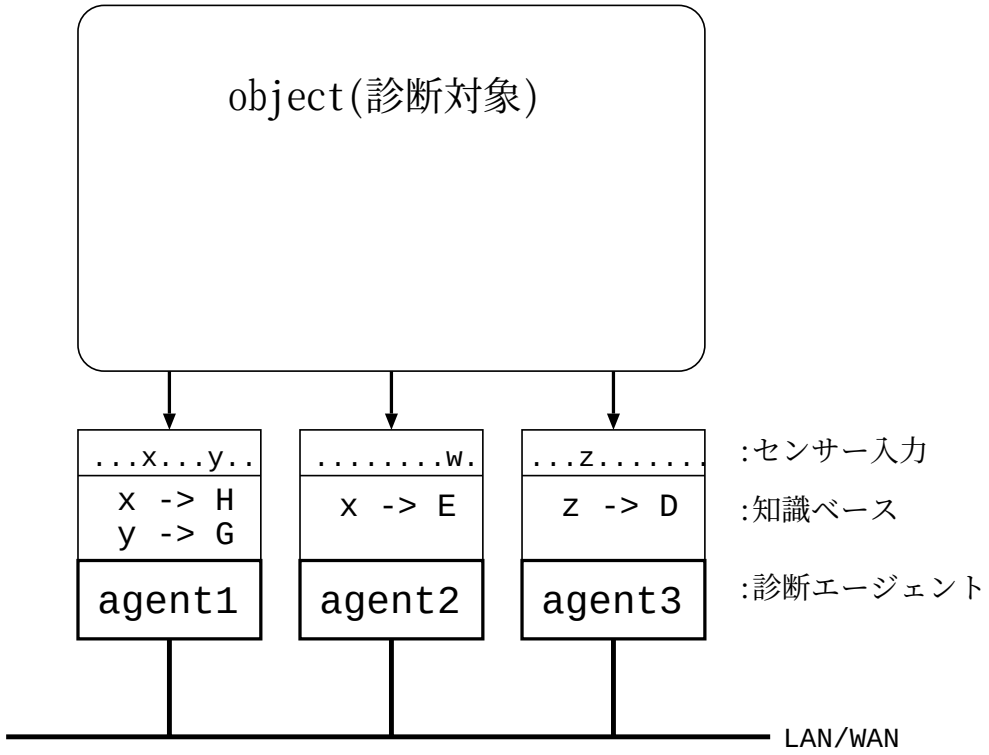

図 2.4: 分散診断 (文献 [1] より転載)

2.3.3 結論と考察

診断エージェントを各センサー毎に分散させて設置することで, あるセンサーやエージェントが故障や保守休止していても, 残されたエージェントで一定精度の診断が実行できることが示されている. このことは, 診断システム全体の頑健性が大幅に向上したことを意味する. また, 各診断エージェントが局所診断することで, 診断システムの仕様変更や拡張に関しては一部のエージェントの知識を入れ換えるだけで対応出来ることになる.

結果として診断対象を構成要素に分割し, 構成要素毎に 1 診断エージェントを設置するという試みは, システムの頑健性と柔軟性の向上をもたらした. しかし, システムの頑健性という観点に立つと情報欠落にのみ対応していれば良いというものではない. 提案する枠組が診断システムである以上, さらなる信頼性向上が求められる. 次節では分散診断システムの頑健性向上の準備として, 分散システムの故障形態について述べる.

2.4 分散システムの故障形態

システムの耐故障性を研究テーマにしたフォールトトレラントの分野においても, 分散システムについて研究されている. そこでは以下の基本問題が扱われている. すべてのプロセッサが同じ値を得る合意 (agreement) 問題, プロセッサ群の中から一つのプロセッサを選択するリーダ選択 (leader selection) 問題, ネットワークの生成木構成 (spanning tree) 問題, 複数プロセッサからの要求のうちの一つに答える相互排除 (mutual exclusion) 問題, 一つのプロセッサから全てのプロセッサへ情報を流す情報散布 (information dissemination) 問題, などである. これら問題についてフォールトトレラントを考えるときに扱われる分散システムの故障形態として, 主に以下の 4 種類が考えられている [3]

停止故障: 故障プロセッサが停止し, メッセージを全く送らない.

送信脱落故障: 送信されるべきメッセージが送信されないことがある故障.

一般脱落故障: 送信, 受理のいずれにおいてもメッセージの脱落がある故障.

ビサンチン故障: 故障プロセッサがどう振る舞うかに仮定をおかない故障. でたらめなメッセージを送ったりする.

ビサンチン故障と脱落故障については分散したプロセッサが全て同じ結果を導く合意問題で扱われる事が多く, それ以外の問題では停止故障のみを扱うものが多い.

上の 4 種類の故障への対応は, 大きく 2 種類に分けて考えられる. それは, 停止故障, 送信脱落故障, 一般脱落故障については情報欠落への対応, ビサンチン故障は誤情報への対応である. 前述のとおり, 情報欠落に対する頑健性はマルチエージェント環境における分散診断システムで保証されているが, 誤情報に関しては対応していない. プロセッサが誤情報を流す可能性を認めるとき, 各エージェントの情報には基本的に誤りの可能性が内在する事になる. つまり, あるエージェントの情報が誤りであることが判明した場合にその情報を推論結果に適切に伝播しなければならないのである. この情報伝播を扱う研究分野に非単調推論がある. 次節ではこの非単調推論について述べる.

2.5 非単調推論

この節では, 非単調推論の概要を述べた後, 本枠組で導入したデフォルト論理 [14] について説明する.

2.5.1 非単調推論

人間の知的活動において不完全な知識は頻繁に用いられる. たとえば, 予測などは不完全な情報から推論する典型的な例といえる. また, あることを正確に推論する際に必要な情報が全て手に入る事はむしろ稀であり, 推論するときは常に不完全な知識を用いると考えてもよい.

ここで従来の論理について考えてみる. 従来の論理では, いかなる時でも正しい結論を導き出す事が目的であるため, 結論に反する可能性が僅かでも残されている場合はその結論に到達しない. しかし, 現実問題として完全な情報を手にすることが期待できない以上, 求められているのはいかなる時でも正しい結論よりも一般的に正しい結論であろう. たとえば「A は鳥である. 」と聞かされた場合に「(一般的に言って)A は飛べるだろう. 」と推論するのは, A がペンギンの時には誤った推論になるにもかかわらず, 妥当である.

先の例で分かるとおり, 不完全な知識を用いた推論には誤る可能性が本質的に内在している. 新しい情報によってある結論が覆された時は, その結論を捨てて正しい結論を導かなければならない. 例では, A がペンギンだと分かった時点で「A は飛べない. 」と推論しなおさなければならない. このように, 知識の増加が必ずしも定理の増加に繋がらない性質を非単調性と呼ぶ. そして, この非単調性を持つ推論が非単調推論である. 非単調推論の代表

的なものには, McDermott と Doyle の NML-I[15], McDermott の NML-II[16], Reiter のデフォルト論理 [14], McCarthy の極小限定 [17] などがある. この節の残りでは, このうちの本枠組で導入する Reiter のデフォルト論理について説明する.

2.5.2 デフォルト論理

デフォルト推論の直観的な解釈は, 「一般的に」または「常識的に」という表現を扱う推論ということになる.

デフォルト規則

デフォルト論理はペア ⟨W,D⟩ で表される. ここで W は公理の集合, Dはデフォルト規則の集合を示す.

$$\alpha : M\beta_1, ...M\beta_n/\gamma$$

ここで, α, β_i, γは自由変数を含む. デフォルト規則は次の直観的解釈を持つ. " もしαを知っていて, 任意の i に対するβ_iが現在の全ての知識に矛盾しなければγを推論する. (最終的には, β_iが無矛盾な仮説だと確かめられる.) $M(\neg CANFLY(x))/\neg CANFLY(x)$ の崩した意味は, "任意の対象に対してそれが飛べないという仮説に矛盾しなければ飛べると仮説する." である.

拡張

デフォルト推論には "拡張" の概念がある. 拡張は出来るだけ多くのデフォルト推論を加えることによって得られる. 拡張の直観的な解釈は, 公理とデフォルト規則から得られる定理の集合ということになる.

$$E_{i+1} = Th(E_i) \cup CGD(D, E_i, E)$$

$$CDG(D, E_i, E) = \{\gamma : (\alpha : M\beta_1, .., M\beta_n/\gamma) \in D, \alpha \in E_i, and\ for\ all\ i, \neg\beta_i \notin E\}$$

ここで Th(X) は X の全ての単調推論の結果を示し, $CGD(D, E_i, E)$ はデフォルト規則と E_i, E から導かれる結果を指す.Eは次のとき$\Delta = \langle W, D\rangle$ の拡張となる.

$$E = \bigcup_{i=0}^{\infty} E_i$$

非単調性

上の定義を用いて, 実際に非単調推論を行う. 鳥の例で考える. 文献 [9] の鳥の例の場合デフォルトの集合は, $D = \{Bird(x) : MFly(x)/Fly(x)\}$ となる. これは, 閉デフォルトではないため以下のような閉デフォルトが無限にあると考える.

$$\frac{Bird(A) : MFly(A)}{Fly(A)}, \frac{Bird(B) : MFly(B)}{Fly(B)}, \ldots$$

推論開始時の公理を $W = \{Bird(Tweety)\}$ とする.

$$E_0 = W = \{Bird(Tweety)\}$$
$$E_1 = Th(E_0) \cup CGD(D, E_0, E)$$
$$= Th(\{Bird(Tweety)\}) \cup \{Fly(Tweety)\}$$

ここで, $Bird(Tweety) : MFly(Tweety)/Fly(Tweety) \in D$かつ $Bird(Tweety) \in E_0$かつ $\neg Fly(Tweety) \notin E$ から, $CGD(D, E_0, E) = \{Fly(Tweety)\}$ が導かれる.

$$E_2 = Th(E_1) \cup CGD(D, E_1, E)$$
$$= Th(Th(\{Bird(Tweety)\}) \cup \{Fly(Tweety)\}) \cup \{Fly(Tweety)\}$$
$$= Th(\{Bird(Tweety), Fly(Tweety)\})$$

$$E_3 = Th(E_2) \cup CGD(D, E_1, E)$$
$$= Th(\{Bird(Tweety), Fly(Tweety)\}) \cup \{Fly(Tweety)\}$$
$$= Th(\{Bird(Tweety), Fly(Tweety)\})$$

$$E_i = Th(\{Bird(Tweety), Fly(Tweety)\})(i \geq 4)$$

従って, $\bigcup_{i=0}^{\infty} E_i = Th(\{Bird(Tweety), Fly(Tweety)\}) = E$ となることから, $Fly(Tweety)$ が導かれる. ではこの時点で$\neg Fly(Tweety)$ が判明したら場合はどうなるだろうか.

$$E_0 = W = \{Bird(Tweety), \neg Fly(Tweety)\}$$
$$E_1 = Th(E_0) \cup CGD(D, E_0, E)$$
$$= Th(\{Bird(Tweety), \neg Fly(Tweety)\}) \cup \{\}$$

ここで, $Bird(Tweety) : MFly(Tweety)/Fly(Tweety) \in D$かつ $Bird(Tweety) \in E_0$かつ $\neg Fly(Tweety) \in E$ から, $CGD(D, E_0, E) = \{\}$ が導かれる.

$$E_2 = Th(E_1) \cup CGD(D, E_1, E)$$
$$= Th(Th(\{Bird(Tweety), \neg Fly(Tweety)\})) \cup \{\}$$
$$= Th(\{Bird(Tweety), \neg Fly(Tweety)\})$$

$$E_i = Th(\{Bird(Tweety)\}, \neg\{Fly(Tweety)\})(i \geq 3)$$

従って, $\bigcup_{i=0}^{\infty} E_i = Th(\{Bird(Tweety), \neg Fly(Tweety)\}) = E$ となるので, $\neg Fly(Tweety)$ が導かれ $Fly(Tweety)$ はもはや導かれない. ここに推論結果が非単調に変化したことになる.

2.6 相互認識ネットワークモデル

分散診断システムにおいて誤情報を取り扱う場合, 診断システム自身の信頼性を知る必要がある. 故障は診断対象だけでなく診断システムにも生じる. 従って診断システム自身が診断されなければならない. 診断システムの診断には注意が必要である. 例えば以下のような犯人探しを考えてみる [23].

犯人探し: 数人の容疑者がいて, それぞれの人が他の人に対して犯人であるないの証言をしたとき, その証言をもとに犯人を探す問題. 図 2 に証言関係を示す.

犯人探しにおいて, テストする側とテストされる側を定義すると以下のようなパラドックスに陥ることが文献 [22] で指摘されている.

> あるシステムを高信頼化するため, そのシステムにその診断システムを付加する. するとその対象システムの故障に関しては信頼化できるが, 診断システムの故障の可能性が残る. その診断システムのまた診断システムを付加すると, またその付加した診断システム自体の故障の可能性は残る. このように, 無限に診断システムのまた診断システムを付加し続けなければならない.

各人がテストし, またテストされることで, このパラドックスは解消される. ユニット U_i がユニット U_j をテストするときその値は以下のように定義される.

$$T_{ij} = \begin{cases} -1, & u_i \text{が犯人でなくかつ} u_j \text{が犯人のとき} \\ 1, & u_i, u_j \text{とも犯人でないとき} \\ 1/-1, & \text{それ以外のとき} \end{cases}$$

このときテスト結果と最も整合する U_i の信頼度は次の動的モデルにより計算される.

$$\frac{dr_i(t)}{dt} = \sum_j T^*_{ji} R_j(t) - \frac{1}{2} \sum_{e_{ij} \in T_i} (T_{ij} + 1), (1)$$

$$R_i(t) = \frac{1}{1 + exp(-r_i(t))}$$

ここで $T^*_{ji} = T_{ij} + T_{ji}$ であり, T_i は u_i によってなされるすべてのテスト集合である.

(1) 式の弟 1 項は, 他のユニットのこのユニットに対する意見を総合したものであり, 弟 2 項はこのユニット自身の他のユニットに対する意見からくる寄与である. 後者は反射効果と呼ばれている.

この反射効果は, あるユニットが信頼度の高いユニットに対し異常であると証言したときにそのユニット自身の評価を下げることに対応している.

$r_i(t)$ は時刻 t におけるユニット u_i の信頼度を反映している. つまり (1) 式の右辺により, ユニット u_i が信頼できるとする評価が信頼できないとする評価を上回ったとき $\frac{dr_i(t)}{dt}$ は正となり, 信頼度が増加する. $R_i(t)$ はこの $r_i(t)$ を [0,1] に正規化したものである. この診断は初期値 $(R_1, R_2, R_3, R_4, R_5) = (1, 1, 1, 1, 1)$ を与えた後に行われる.

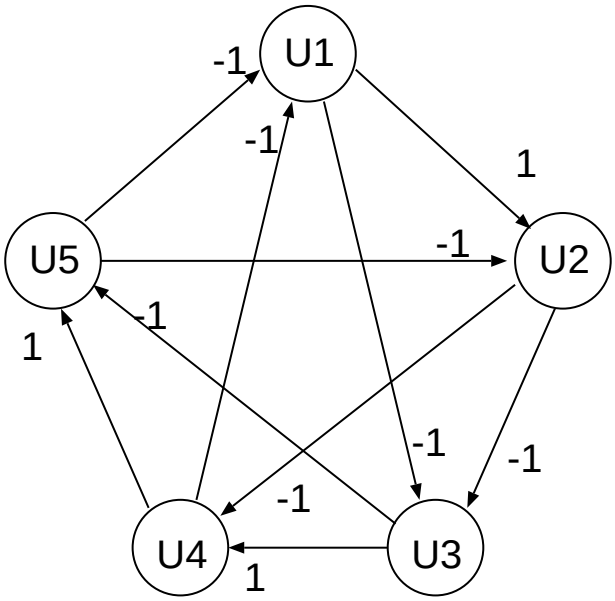

図 2.5: 証言関係 (文献 [22] より転載)

第 3 章

非単調推論を用いた分散診断システム

本章では, 本研究で提案する「非単調推論を用いた分散診断システム」のモデル化を行う. 章構成は, 前半で文献 [1] に習う点と新たな提案をはっきりさせ, そのあと各種エージェントの定義, エージェント間通信の手順等を説明する.

3.1 システム概要

この節の目的は, 本研究の基となる文献 [1] の定義に習う点と, 本研究で新たに提案する点を明確にすることである.

従来の枠組に習う点

以下に示す点は, 文献 [1] の定義に習う事とする.

- 診断対象は, プラント, 分散 OS, 通信システムなど, 個々の構成要素が分散していて, 全体として 1 つのシステムをなしているものとする.
- 診断対象をセンサーで監視する. センサーからは部分的な情報しか得られない.
- 各センサーには, プロセッサが接続されていて, センサー入力をローカルに保持している.
- 各プロセッサ上には, 診断エージェントが動作している. 各診断エージェントは各々独自の知識ベースを持ち, 局所診断している.

- 診断モデル. 診断モデルは, 対象の設計情報を基に作成された, 根が異常原因, ノードが中間現象, 葉が異常兆候を示す AND 木である.

- 各エージェントは, 複数の診断モデルを持つ事が出来る. 各エージェントの持つ診断モデルは各々独自なものであって良いので, 局所診断に必要な診断モデルを中心に保持できる. また, 診断対象の部分的な変更に対して, 一部のエージェントの知識をいじるだけで対応できることから, システムの柔軟性が期待できる.

- 事象駆動型仮説推論. 異常原因推論は, ある診断エージェントが異常兆候を検出した後にはじまる. 推論形式は仮説推論で, 自身の担当以外の現象を仮説し推論を進めて行く. 仮説された現象の成否は他のエージェントの判断に委ねる事になる.

新たな提案

以下に示すものは本枠組で提案したものである.

- 診断エージェント間の通信は黒板と呼ばれる共通メモリを通して行われる. ただし黒板は, 推論, 判断などの一切の処理を行わない. 黒板定義のためボトルネックに陥る可能性があるが, 本枠組では黒板を用いることによる推論の収束性を重視し導入した. また, 各エージェントが黒板をポーリングすることで, 全く知識の無いエージェントとも通信する事が出来る.

- 各診断エージェントは互いに診断し, 診断される相互診断の形をとっている. 従って, 診断システム付加のパラドックス (診断システムの診断システムを無限に定義しなければならないという) に陥ることなくシステムの信頼度を高める事が出来る.

- 異常原因推論に非単調推論を導入したこと. 本枠組ではシステムの頑健性を目指すために, 診断エージェントが誤情報を流す可能性を認めている. 従って, エージェントの情報の誤りに対応して推論結果を修正する機構, 非単調推論が必要となる. 本枠組ではデフォルト推論を用いた.

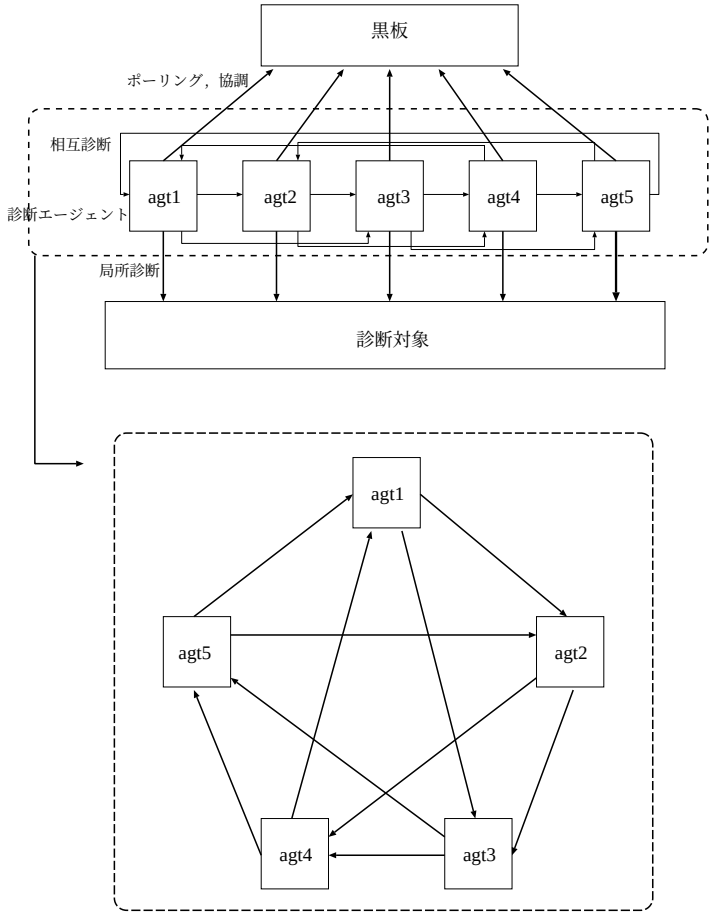

図 3.1: システム概略図

3.2 診断モデル

診断モデルは, 文献 [1] に習い, 根が異常原因, ノードがその異常が原因で引き起こされる中間現象, 葉が最終的にセンサー等に現れる異常兆候を示す. これら現象の因果関係は診断の完全性を考えて, システムの設計情報から抽出されたものとする. また, 木は AND 木とし各診断エージェントが持つ木は部分的なものとする. 診断モデルを持つエージェントの担当範囲を超えた現象は仮説という形で木のノードに納められる. つまり, 各診断エージェントは検出された異常兆候から異常原因を推論するために, どの異常兆候もしくは中間現象を仮説すれば良いかを知っている.

診断モデルの例を図 3.2 に示す. 図の矢印の左側が完成したモデルを示し, 右側が各診断エージェントの持つ部分的なモデルを示す. 点線で書かれた枝の先にある現象は仮説を表

す. 図を用いてエージェントの仮説推論過程を簡単に説明すると, 診断モデル S1-X4 を持つエージェントが異常兆候 “out16=1” を検出したとする. このときエージェントは診断モデルを用いて異常兆候 “out9=1”,“out17=1” を仮説し, 異常原因 “S1-X4” を推論する. (ただし, 診断モデルが AND 木であるため, 異常兆候 “out9=1” と “out17=1” の両方の否定情報が無い場合に限りこの推論は成立する.) ここで仮説された異常兆候の成否は, その兆候を検出できるエージェントの判断に委ねる事になる.

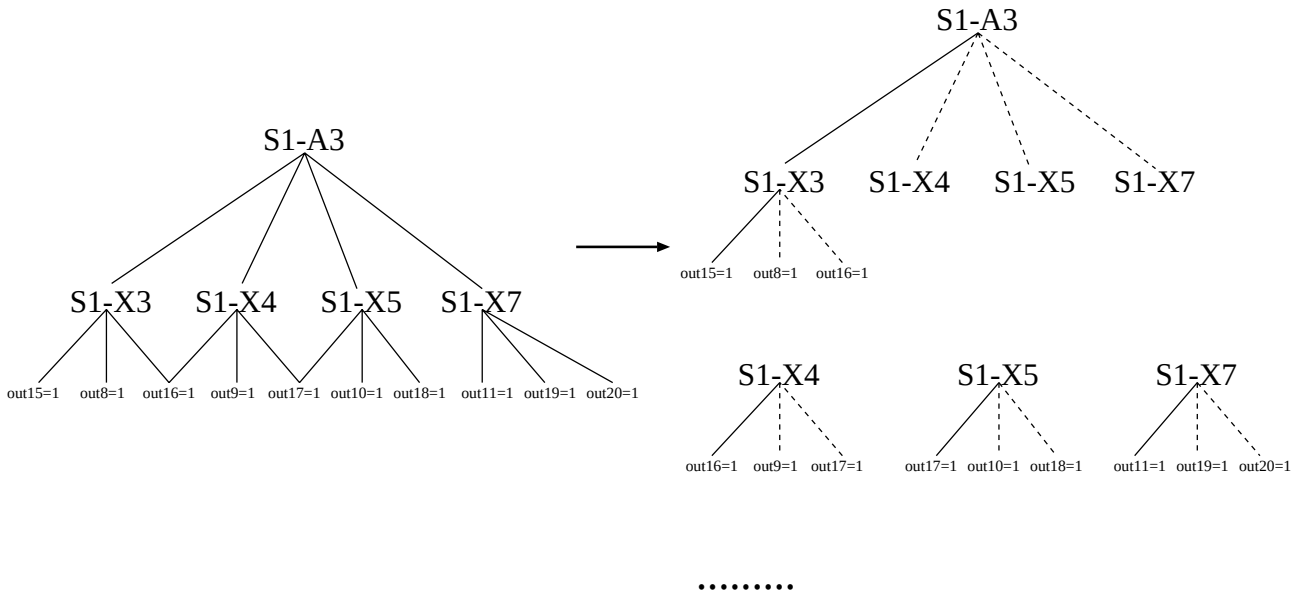

図 3.2: 診断モデル

3.3 システム構成要素の定義

3.3.1 診断エージェント

本研究では, 診断対象の各構成要素毎に 1 エージェントが診断を担当する. 各エージェントはセンサーから得られた情報をもとに異常兆候を検出し, 診断モデルを用いて出来る限りの推論を行う. また, エージェント間で互いに正常に動いているかをテストする. 診断エージェントの持つ知識として, センサー情報から異常現象を読み取る知識, 異常現象から中間現象, 異常原因を推論するための診断モデル, そして他のエージェントをテストする知識の

3 種類を定義している.

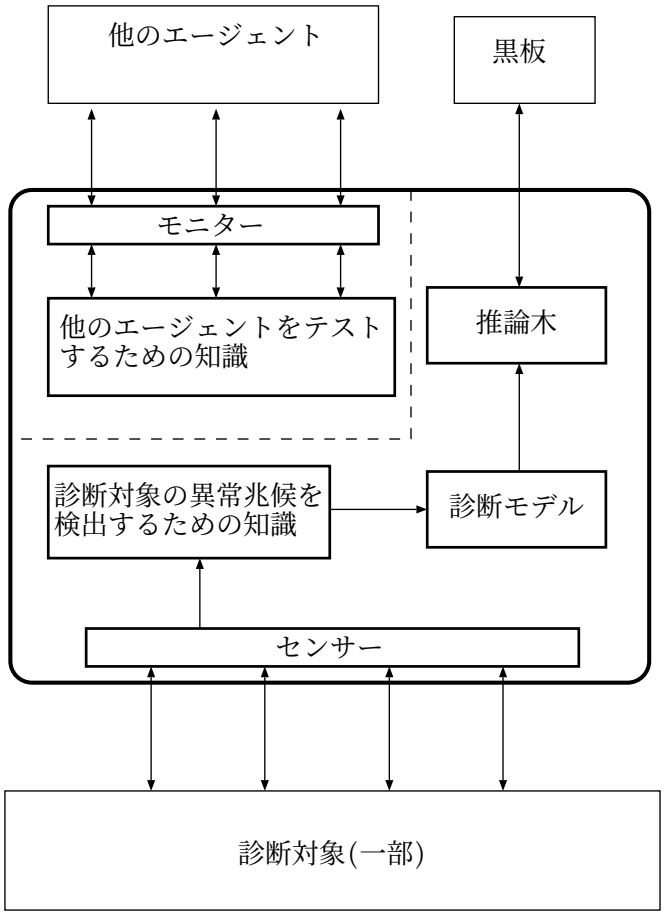

図 3.3: エージェントの構成

3.3.2 黒板の構成

エージェント間通信は黒板と呼ばれる通信媒体が用いられる. 黒板の構成は, 異常兆候を書く phen リストと, そこから推測される原因木, 中間現象木を書くリスト, そして相互診断の結果を書くリストからなる. 相互診断結果を除く各リストは In リストと Out リストに分かれ, 診断エージェントの情報が誤情報であったときに備える. ここで, In リストには葉ノードを否定されていない木が入り, Out リストには, 葉ノードが否定された木が入る. また, 黒板上の木の葉 (異常兆候) には各エージェントの信頼度が付けられていて, 信頼度が 0 のときに限りそのエージェントの情報は無いもの (不定) とされる.

図 3.4: 黒板の構成

3.4 エージェント間通信

この節では, 実際の診断がどのような手順で行われるか, またその際のエージェント間通信はどう行われるかについて説明する.

1. 診断初期

本枠組において各診断エージェントは, 平時は, 担当する構成要素の局所診断と黒板に何か書き込まれていないかを定期的に調べるポーリングを行っている. システムにある異常が生じると, 診断は以下のようにして始められる.

1. エージェントはセンサーから定期的に送られて来る情報を解析している. このときセンサー情報から異常兆候が検出されるとシステムの診断が開始される.

2. センサー情報を解析して得られた異常兆候と, エージェント自身のもつ診断モデルを用いて, その異常兆候を説明する原因を推論する. このときに確かな情報は今得られた兆候だけなため, 生成される推論木は複数存在する. (もちろん単一の時もある.)

3. 推論木が完成すると, その情報を持って黒板を見に行く. ここで黒板に何も書き込まれていない場合は, 相互診断のプロセスを走らせる.

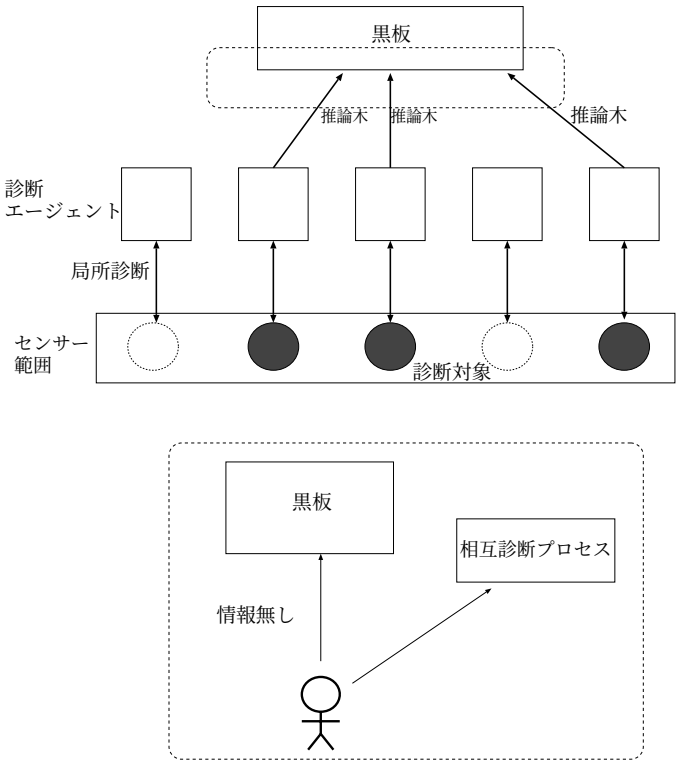

図 3.5: 相互診断開始

2. 黒板上の処理

異常検出時: 1. 黒板に到達したらまず黒板を見て, 今持っている不完全な推論木の仮説部分を埋めてくれる情報を探す. その情報があれば推論木を更新する.

2. 次に黒板上に書かれている木の仮説に目を向ける. エージェントが検出した現象が仮説されていたら, その仮説の成立を告げる.

3. 黒板上の phen リストに, 今検出した現象を付け加える.

4. 黒板上の tree リストを見て, 今持っている木がまだ報告されてないとき, その木を tree リストに加える.

ポーリング時: 1. 黒板上に書かれている木の仮説に目を向ける. エージェントの担当範囲の現象が仮説されていたら, その成否を告げる. このとき, 必要ならばIn リスト-Out リスト変換, Out リスト-In リスト変換を行う.

2. 黒板上の phen リストに検出した現象が無い場合, その現象を付け加える. このとき, 仮説の否定を検出すれば Out リストに, 逆に仮説の成立を検出すればIn リストに付け加える.

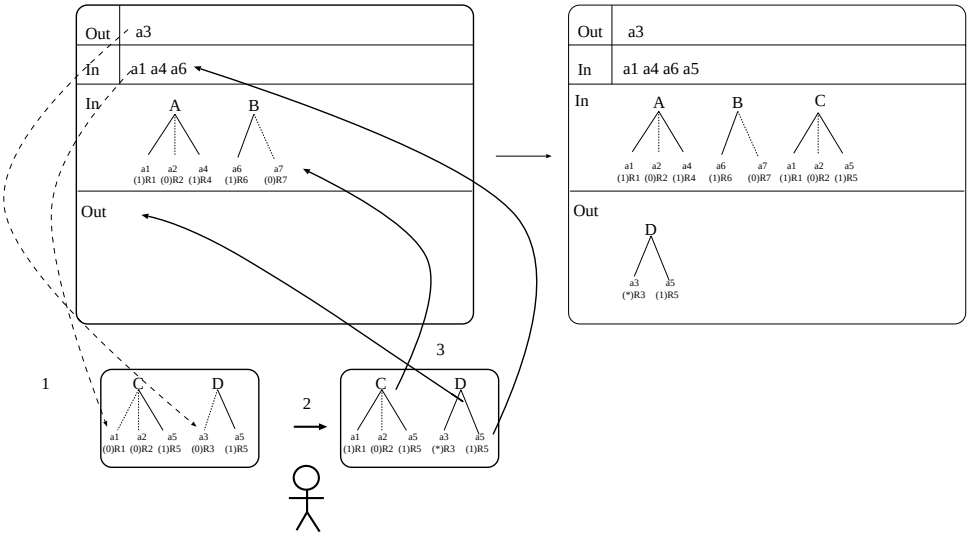

図 3.6: 黒板上の処理

3. 診断の終了時期

分散診断システムにおいて診断終了の時期を見定めるのは難しい. その理由は幾つか考えられが, 一番重大な理由は通信路の状況に依存するメッセージ遅延の可能性である.

分散システムにおける診断エージェント間の協調は, 全てメッセージ通信によって行われる. 従って, 通信路の太さや通信量の関係でメッセージの遅延の可能性を内在している. 診断において局所的な視点での推論が, ある追加情報によって覆されることは多い. 検出された情報を最も良く説明する原因を推論するためにはより沢山の情報が欲しいが, メッセージ遅延とプロセッサ停止故障を区別するのが事実上不可能である以上, 情報をいつまで持てば良いかという問に対する明確な解答は無い.

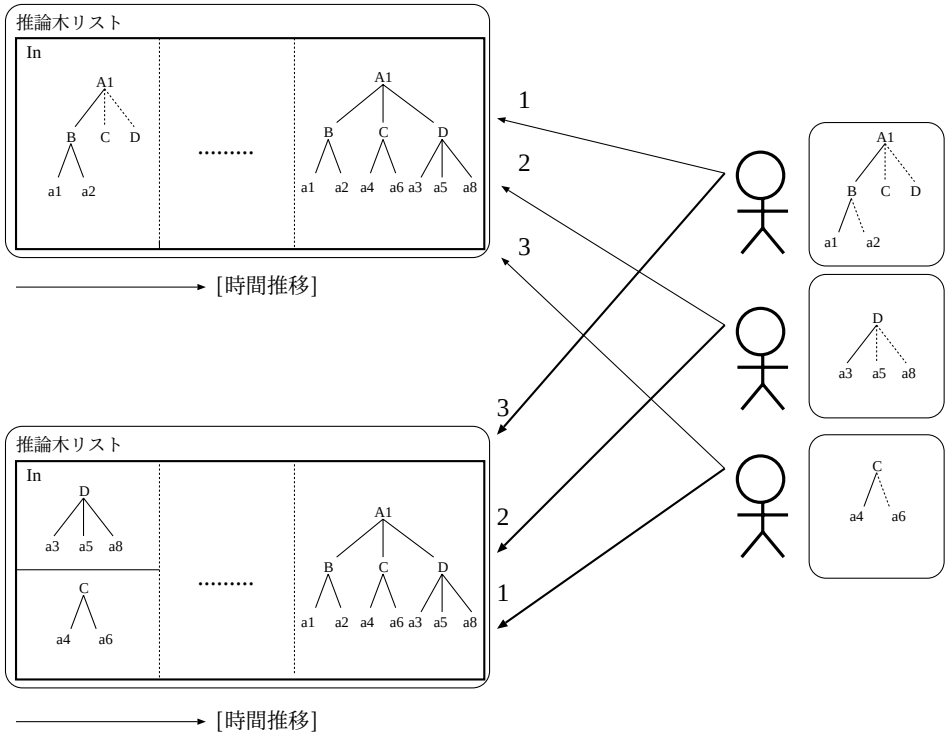

図 3.7: 非同期性による推論過程の違い

図 3.7 は, 診断エージェントの非同期性による診断過程の違いを示している. 上側の図の順番で黒板が更新された場合, 推論過程を通じて異常原因は “A1” である. しかし, 下側の図の順序で更新された場合, 診断前半では異常原因は “C” と “D” という推論結果になっている. この場合正しい推論結果を得るためには, エージェント 1 の到着を待たなければならない.

以上の理由から本枠組では診断終了をユーザの手に委ねることにする. 診断はユーザが終了の合図を送るまで続けられ, 定期的に診断結果を出力することにする.

3.4.1 信頼度付きのエージェントの協調

故障エージェントの誤情報に対して, 頑健性を保つために診断エージェントの信頼度を計算する. 信頼度計算は以下の手順でなされる.

1. 異常兆候が検出され, 診断システムが動き出すと同時に相互認識ネットワークによってエージェントの相互診断を行う.

2. 診断結果は黒板に書かれる. 信頼度は R_iで示され, $R_i = 1$ のときエージェント i は信頼でき, $R_i = 0$ なら信頼されない事とする.

3. $R_i = 1$ のとき, エージェント i の観測は公理となり, その後覆される事は無くなる. また $R_i = 0$ のとき, エージェント i の観測は全て無視される. (現象の成立, 不成立が決定されていない状態を “不定” と呼ぶことにする. この定義に従えば $R_i = 0$ のときエージェント i の情報は全て不定になると言える.)

黒板上の表現で, 公理化を表すと図3.8のようになる. 各葉に対する評価のところの $(1)R_i$(検出), $(0)R_i$(不定), $(*)R_i$(不成立) の表現は, これらの評価は “覆される” 可能性があることを示す. しかし, 相互診断の結果各診断エージェントの信頼度が判明すると, 各葉に対する評価は公理となり 1(検出), 0(不定), *(不成立) のように表現される.

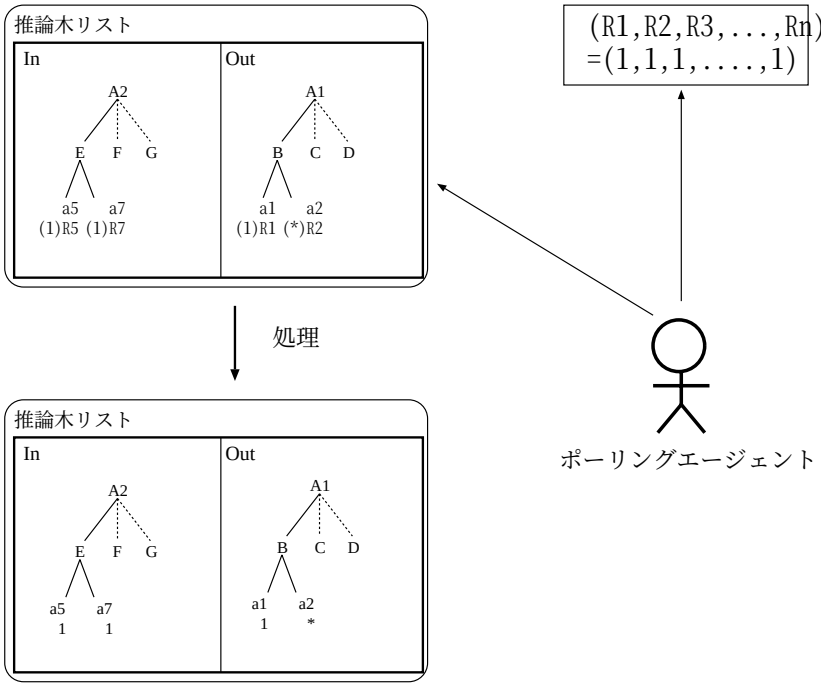

図 3.8: 検出現象と信頼度

3.5 非単調推論の定式化

黒板上の推論木 (In リスト) を Reiter のデフォルト論理によって定式化する.

デフォルト規則: デフォルト規則を以下のように定義する.

$$\frac{d - agt(x_i) : MR_i = 1}{O(x_i)}$$

ここで, d-agt(x) は自由変数 x が診断エージェントであることを示し, O(x) は x の観測を示す. 直観的解釈は, x が診断エージェントであり, かつその信頼度が 1 で無い事が証明されなければ観測 O(x) が導かれる. 推論はデフォルトの集合

$$D = \{d : d = \frac{d - agt(agt_i) : MR_i = 1}{O(agt_i)} (1 \leq i \leq 21)\}$$

を与えられた後に始まる. この枠組では, デフォルト論理の公理に当てはまるものがない. 従って以下の規則を付け加える.

$$d - agt(x_i) \wedge R_i = 1 \rightarrow O(x_i) \subset W$$

拡張: 黒板上の推論木 (In リスト) はデフォルト論理の拡張と同等である. 従って診断結果 E はデフォルトの拡張の定義に習い以下のように定義される.

$$E = \bigcup_{i=0}^{\infty} E_i$$
$$E_0 = W$$
$$E_{i=1} = Th(E_i) \cap CGD(D, E_i, E)$$
$$CGD = \{\gamma : (\alpha : M\beta_1, .., M\beta_n/\gamma) \in D, \alpha \in E_i, and\ for\ all\ i, \neg\beta_i \notin E\}$$

実際のシステム動きを例示すると以下のようになる (図 3.9). 図は, エージェント 2 の情報によって Out リストに入っていた推論木 “A1” が, 相互診断の結果エージェント 2 の故障が判明し, In リストに復帰する様子を表している.

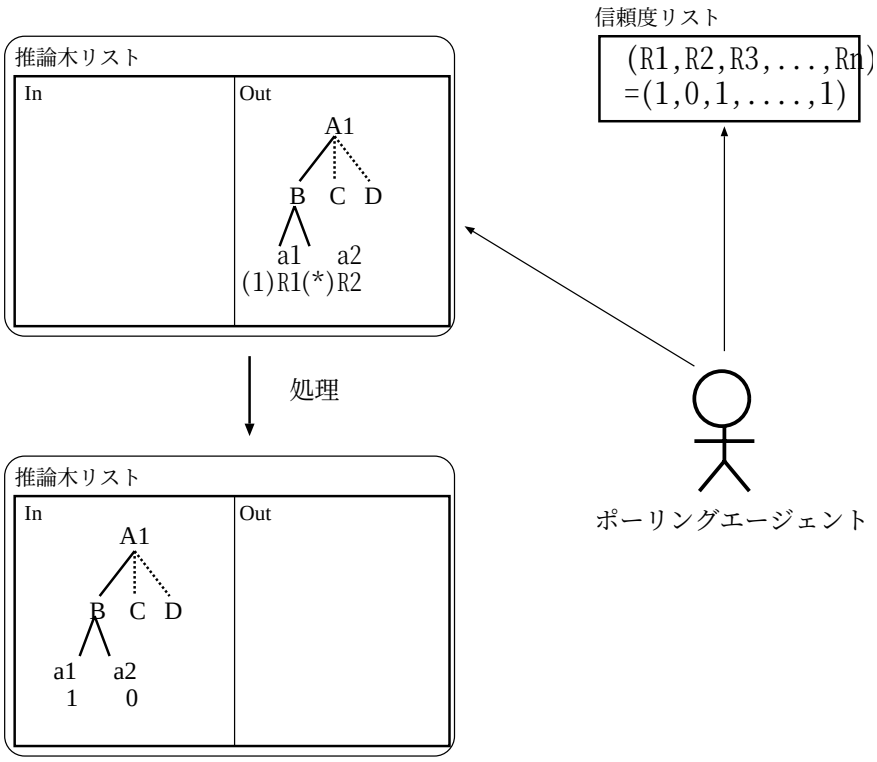
信頼度リスト
(R1,R2,R3,...,Rn)
=(1,0,1,....,1)
推論木リスト
In
Out
A1
B C D
a1 a2
(1)R1(*)R2
処理
推論木リスト
In
Out
A1
B C D
a1 a2
1 0
ポーリングエージェント

図 3.9: 非単調性

第 4 章

実験

本章では, 前章でモデル化した診断システムを, 現在東京大学米澤研究室で開発中の並列オブジェクト指向言語 ABCL/f を用いて並列プロセッサ AP1000 上に実装し, 各種機能の実証を行う. 章構成としては, 前半で ABCL/f の概要, 診断対象の説明を行い, その後実験内容, 最後に結果と考察を述べる.

4.1 実験環境

この節では実装に用いた言語の説明と, プログラムの説明, 診断対象モデルについて述べる.

4.1.1 並列オブジェクト指向言語 ABCL/f

実装に用いた ABCL/f の概要を述べる. [24].

future の導入

基本概念は Halstead により提案された. もともとの future は暗黙的に同期をとるものであったが, ABCL/f では明示的に同期処理を行える. future 起動の際には, 通常の引数に加えて future オブジェクトと呼ばれる返答届け先を生成する. 処理結果はこの future オブジェクトを経て伝達される. また, この future オブジェクトは一階の値として扱える.

デフォルト規則の提供

ABCL/f はデータ配置と計算処理の並列性がプログラマにとって可視であることを基本としている. これは, 全ての処理をプログラマに委ねるということではない. 特に指定が無ければ, コンパイラはデフォルト規則にしたがい処理し, プログラマはこのデフォルト規則から明示的に逃れる方法を合わせ持つということである. 例えば, プロシージャは指定が無ければローカルノードで呼び出されるが, プログラマは付加的な指示を用いて明示的に指定したノードで呼び出す事も可能である.

これら future とデフォルト規則によってプログラマは非同期起動を基本とする多くの並列プログラムを記述する際に, 逐次版に大きな変更を加える必要が無くなる. このことは, プログラム開発時に逐次版から着手し, 段階的に改良しながら開発をすすめることが可能であることを示している.

更に以下の並列オブジェクト指向言語としての性質を合わせ持つことから, 本枠組の実装に ABCL/f を選択した事は妥当だと思われる.

- システムのオブジェクト単位で情報隠蔽されていて, メッセージ通信のみによりその値の参照, 書き換えが可能である.
- 並列の単位がオブジェクトである.

4.1.2 プログラムの説明

診断対象

図 4.1 は診断対象の論理回路である. 診断は入力値を全て 0 にして行われる. 論理回路は構成要素の異常伝播を比較的容易にシミュレートでき, また, 設計情報からシステムの動作を完全に把握できるなど分散診断システムを試すために必要な性質を持つ事から診断対象としてふさわしいと判断した. 論理回路の異常伝播の例を挙げると, 図 4.1 の論理回路の入力値が全て 0 でかつ AND ゲート 1(図中 A1) の出力線が 1-縮退故障していると仮定すると, その影響 (ゲートの出力が 1 になる) は, X1, X2, X8, X9, O7, O8 にまで伝播する. 各診断エージェントはそれぞれ論理ゲート診断を担当する. 担当エージェント番号は, 各論理ゲートの出力線の番号と一致する. 例えばエージェント 21 は出力線が “out21” である OR ゲート O8 を担当する.

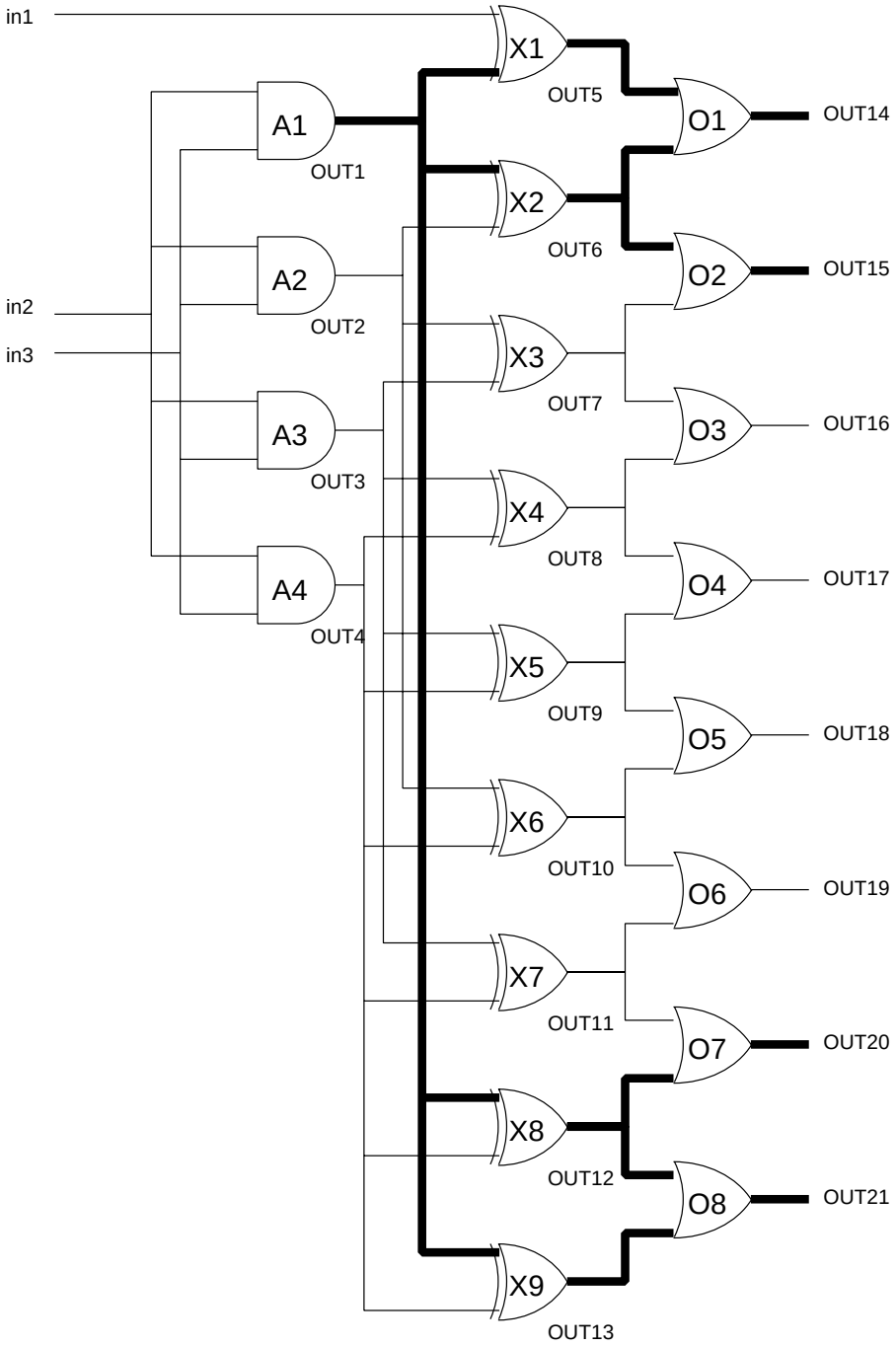
in1
in2
in3
A1
A2
A3
A4
OUT1
OUT2
OUT3
OUT4
X1
X2
X3
X4
X5
X6
X7
X8
X9
OUT5
OUT6
OUT7
OUT8
OUT9
OUT10
OUT11
OUT12
OUT13
O1
O2
O3
O4
O5
O6
O7
O8
OUT14
OUT15
OUT16
OUT17
OUT18
OUT19
OUT20
OUT21

図 4.1: 診断対象回路

診断モデル

各エージェントの持つ診断モデルを図 4.2 に示す.

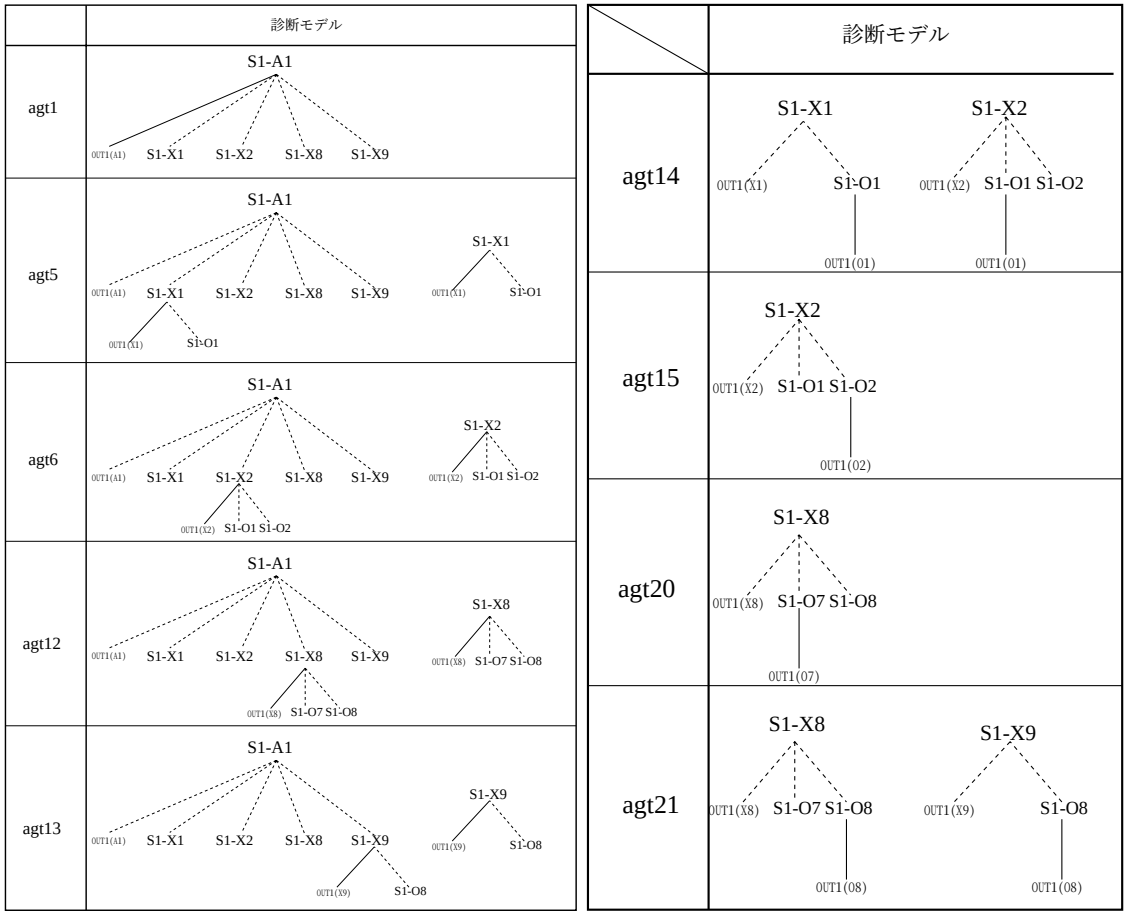

図 4.2: 診断モデル (実験用)

エージェントの構成

ここでは, 前節で定義したエージェントを実際どのように実装したかについて述べる. エージェントの内部変数の定義を以下に示す.

```
(defclass agent ()
  ((list (list tree)) static-tree)
  ((list tree) detect-tree)
```

```
((list sample) sam)
(nameconf nconf)
(point bp)
(fixnum kind)
((list (list fixnum)) know)
)
```

各内部変数の説明

static-tree: 診断モデルにあたる. ある異常兆候から仮説推論される診断モデルをリストとして保持しているため, エージェントが検出出来る異常兆候分のリストを持つ. 従って型は診断モデルリストのリストになる. 検出された異常兆候に反応し, その兆候を含むリストが発火される.

detect-tree: static-tree に異常兆候が入力されたとき発火されるリストがここに入る. 黒板に持って行って処理するのに使われる.

sam: 相互診断用の知識である. 型 "sample" は相互診断時の通信相手アドレスとその相手に対する評価 (テスト結果) のセットである.

nconf: エージェント自身の名前と, 信頼度のセット. 信頼度は相互診断前 (つまりデフォルト値) は 1 である.

bp: 協調が必要な診断時の通信相手 (黒板) のアドレス. 全てのエージェントに共通.

kind: エージェントが担当する診断対象の構成要素の種類. 具体的には, AND, OR, EXOR というゲートの種類が入る. 相互診断時にこのエージェントをテストするエージェントが用いる.

know: 相互診断時に用いる知識. 相手の担当ゲートの種類に合わせたもの. 具体的には各種ゲートの真理値表.

黒板の構成

黒板の内部変数を以下に示す.

```
(defclass blackboard ()
  (st-list d-phen)
  (st-list treelist)
  (fixnum hist))
```

各内部変数の説明

d-phen: 各診断エージェントが報告して来た異常兆候を保管する場所. In, Out に分かれている.

treelist: 各診断エージェントが推論した異常原因木を保管する場所. In, Out に分かれる.

通信用メソッド

エージェント-黒板間

renewal-tree: 黒板の情報を基にエージェントが推論した診断モデルを更新するメソッド.

renewal-bbtree: エージェントの持つ情報を基に黒板上の木を更新するメソッド. renewal-tree の後に実行される.

bb-diag: エージェントが持つ推論木による黒板上の処理を行う. renewal-bbtree もこのメソッド内で行われる. エージェントが持つ情報が新しい場合は黒板に書き足す等の処理も行う.

polling: エージェントが定期的に黒板を見に行くもの. 黒板上の木をみて, 自身の担当範囲の仮説がある場合はその成否を知らせる.

reset-bbtree: 黒板上の木をエージェント情報に基づいて更新した際に, In-Out 変換または Out-In 変換が生じる可能性がある. それら変換を行うメソッドである.

エージェント間

text: 相手エージェントをテストする関数. 相手の診断対象の種類を判別し, それに合わせてテストする. 例えば, 相手が AND ゲートを担当していた場合, AND ゲートの真理値表に合うかどうかを調べる.

agt-test: 前述の text 関数の帰り値に応じて, その相手エージェントを評価するメソッド. この評価値を用いて相互診断を行う.

エージェント分散化

並列オブジェクト指向言語 ABCL/f では, 1 プロセッサ 1 エージェントのマルチエージェント環境を実現してくれる. 具体的処理は, エージェントの内部変数とメソッドを各々任意のプロセッサに割り当てるだけで良い. ABCL/f において, あるオブジェクトのメソッドが呼び出されると, そのメソッドはエージェントが割り振られているプロセッサで処理される. 従って完全な分散環境が実現できる.

実行例

システムの実行コマンドは以下のようになる.

```
./main-agt.sun s1-a1 agt1
```

引き数の意味は次のようになっている.

- 第一引数で, 診断対象の故障原因を指定する. s1-a1 は回路中の論理ゲート A1 の出力線が 1-縮退故障している事を示す. 故障の種類は s1-a1, s1-a2, s1-a3, s1-a4 の 4 種類を用意した.
- 第二引数以降では, 指定された診断エージェントが故障していることを示す. agt1 はエージェント 1 が故障していることを意味する. agt1-21 全てを故障させる事も出来る.

以下は出力例である.

```
([S1(X8)]
     ([S1(O8)]
     ([out21=1 3])
)
     ([S1(O7)]
     ([out20=1 0])
)
```

```
    ([out12=1 1])
)
```

この場合, S1-X8(論理ゲート X8 の縮退故障) が生じていることになる. “([out20=1 1])” において, 右側の数字が 1 なら, その現象が検出された事を示す. 他, 3 不成立, 0 不定を各々示す.

4.2 実験

4.2.1 処理の効率化

まず実験環境について述べる. 先にのべた回路の診断で, あるエージェントが異常兆候を検出してから, 全てのエージェントが局所診断を終えるまでの処理時間につい p ての測定を行った. 診断エージェントの数は論理ゲート数と同じ 21 で, この 21 のエージェントを割り当てるプロセッサ台数を増やす事で処理の並列化をはかった. また 1 エージェントの負荷の時間推移についての実験を行い, 平時と協調診断時の負荷の差を測定した.

(1). 1 エージェントの負荷の時間推移

この実験では各エージェントのポーリングが開始されてからしばらく後に診断対象が故障するようにした. 図 4.3の横軸は時間推移 (ここでは, 各エージェントに 40 回ずつポーリングさせているため, そのポーリング番号を時間推移とした.) 縦軸はその時点 (ポーリングステップ) でのエージェントの負荷を表す. エージェントの負荷は, 黒板を見に行き必要なら書き換えを行うプロセスの実行時間で表されている.

図 4.4は, 診断に関係した 3 エージェントの負荷の時間推移を示している. 診断対象に故障が生じるまでは, 各エージェントに負荷が殆んど無い. これは, プログラム実装の簡単のために診断対象の平時の局所診断を単純なものにしたためで, 実際はいくらかの負荷がかかる. 2-5 ステップ付近では各エージェントの負荷が増大している. このとき黒板を用いた協調診断が行われていると予想される. そして, 推論の終了とともに負荷も大幅に減少する.

次に協調に参加しないエージェントについて調べる. 尚, 先程の図と比べて, グラフの縦軸の区切りが異なっているいる点に注意が必要. 図 4.4は, 図 4.3と同じ実行内でとられた協調診断に無関係なエージェントの負荷の時間推移である. ポーリングステップ 4 あたり

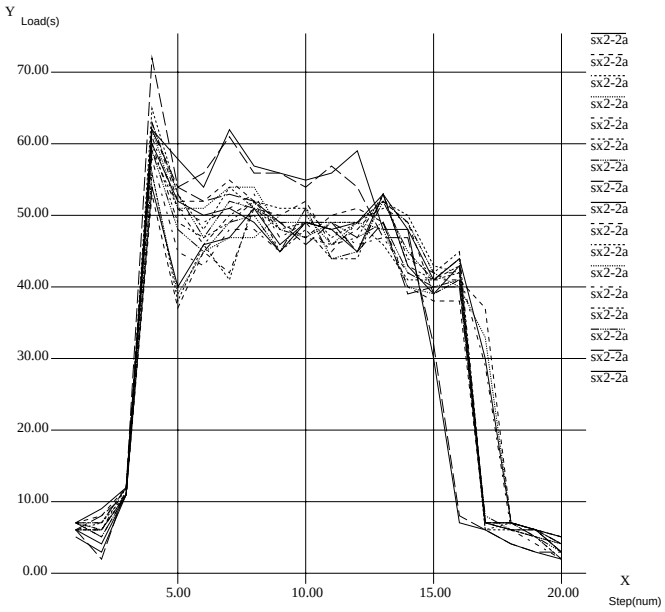

図 4.3: agent の負荷の時間推移 (協調診断)

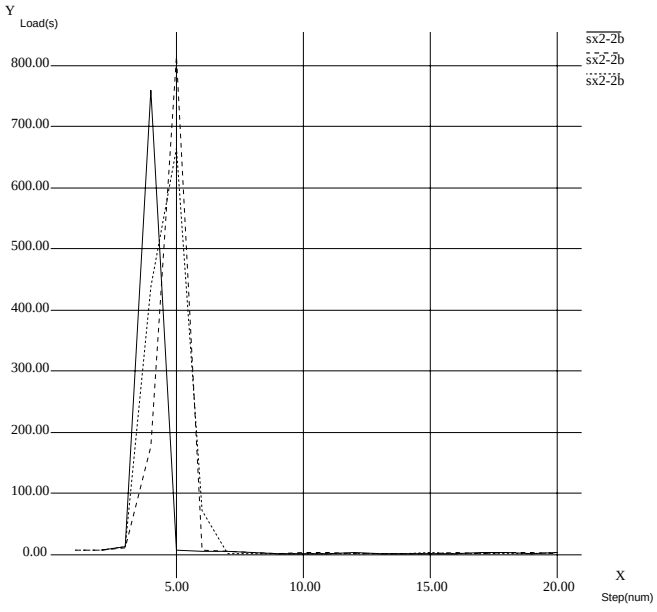

図 4.4: agent の負荷の時間推移 (ポーリング)

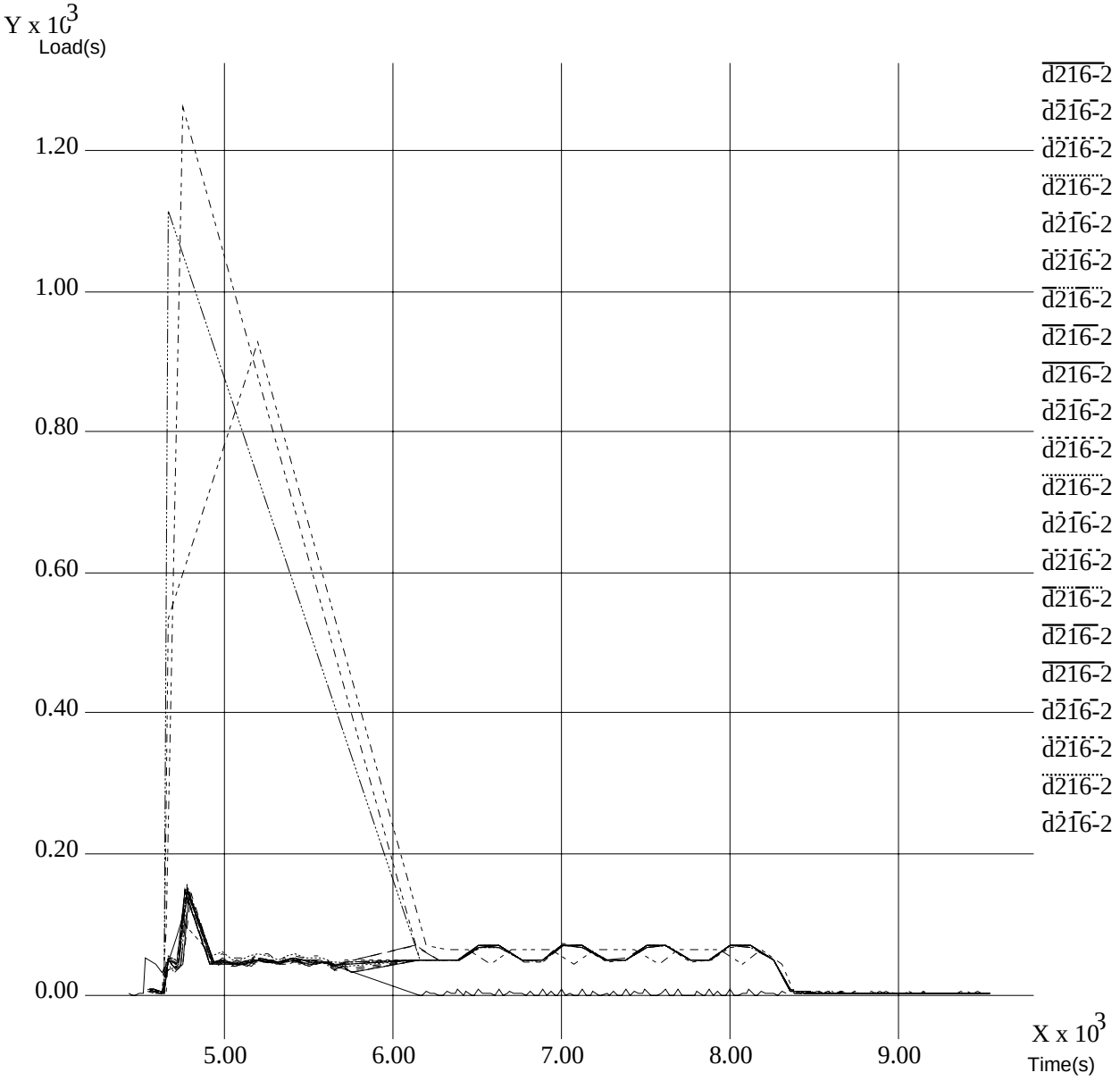

図 4.5: agent の負荷の時間推移 (全 agent)

で負荷がかかり, その後は少しずつ負荷の減少が見られる. ステップ 17 あたりで負荷が再び失われている. つまり, 黒板上に木がある状態ではポーリングによる負荷がかかっている事を示す. また, 黒板を用いた協調が行われている場合, その通信量などの関係から負荷がやや増加する. 従って, 図 4.4は, ポーリングステップ 2 から黒板の協調が始まり, ステップ 18 付近で推論を終了し黒板がクリアされたことを表している.

図 4.5は, 横軸に実時間をとったものである. この図から想像される, エージェントの推論過程は以下のようになる.

4.5(s) まで: 局所診断を表す. 各エージェントは診断対象の局所診断と黒板のポーリングを行っている. このとき黒板には何も書かれていないため, 負荷も小さい.

4.5-6.2(s): 診断対象に故障が発生し, 故障原因を推論している過程を表す. このとき協調しているエージェントは図 4.3の 3 個体である.

6.2-8.4(s): 担当範囲に異常が無いエージェントのポーリングを表す. 黒板に木があるときのみ負荷が増える. 従ってこの区間は黒板上に木が存在していることを示す.

8.4(s) 以降: 診断終了を表す.

(2). システムの並列性の実験

実験結果の図の簡単な説明をする. このグラフの横軸は使用したプロセッサ台数, 縦軸は処理時間を表している. 図中のグラフで “handX” とあるがこの添字 X が大きい程, 各エージェントの局所推論の負荷が大きいことを示す.

通信コストの影響

図 4.6から, 各エージェントの局所推論の負荷が軽いときは単一プロセッサで処理したほうが効率的であることが分かる. 理由は明白で, 処理を分散化した時に加算される通信コストが, 全体の処理時間と比較してかなり大きいことである. このことは, グラフが不安定であることからも見てとれる. 通信コストは通進路の状況によって変化するものであり, かつ処理時間全体に与える影響が大きいため, この通信コストの変化が全体の処理時間を不安定にする原因となっているのである.

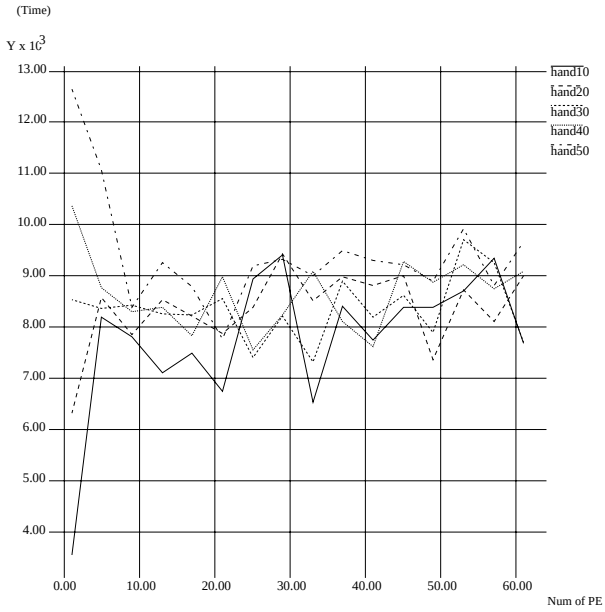

図 4.6: 並列化による処理の効率性 (1)

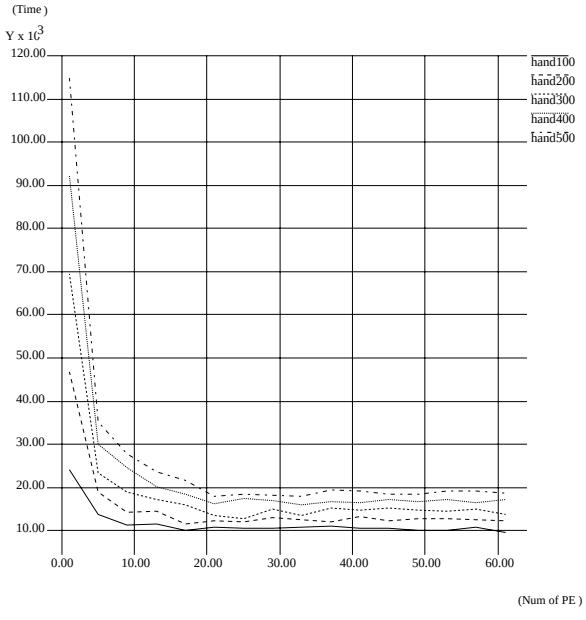

図 4.7: 並列化による処理の効率性 (2)

並列処理効率

局所推論の負荷が一定限度を超えると, 並列処理による効率化が見られる. これは図 4.7 は見れば明らかで, 全体の処理時間が多くなればなるほど, 並列処理による効率化がなされることが見てとれる. また, 図 4.7ではプロセッサ数が 21 を超えるとそれ以上の効率化がされなくなる. これはエージェント数の 21 と一致することから, 各並列要素にかかる負荷が均等になるよう設計すればそれに見合った効率化が得られることを示している.

4.2.2 頑健性の実験

ビサンチン故障

21 の診断エージェント中で誤情報を流すエージェント数を変化させて診断を行った. 結果を表 1,2,3 に示す. 表中で a1, a2,... とあるがこれは, エージェント 1, エージェント 2,... を, また no.1, no.2,... は実験番号を各々示している. また, 表中の “◯”, “×” はエージェントの状態を示し, ◯は正常, ×は故障を表す. 尚, 不定状態と停止状態は本質的に同じであるため, この実験は停止故障に関する実験を含んでいる.

1 診断モデル複数エージェント

表から, 故障エージェントの情報が不定とされ, 無視されている様子が見てとれる. 故障エージェント数が 1 の時と, 2 の時は診断への影響は故障エージェントの情報が欠けるだけである. この実験では同じ診断モデルを複数のエージェントに持たしているため, 1,2 個のエージェントの故障に対しては, 高い診断精度を保つ事が出来る.

	a1	a5	a6	a12	a13	a14	a15	a20	a21	結果
no.1	×	○	○	○	○	○	○	○	○	S1(A1)(8/9)
no.2	○	×	○	○	○	○	○	○	○	S1(A1)(8/9)
no.3	○	○	×	○	○	○	○	○	○	S1(A1)(8/9)
no.4	○	○	○	×	○	○	○	○	○	S1(A1)(8/9)
no.5	○	○	○	○	×	○	○	○	○	S1(A1)(8/9)
no.6	○	○	○	○	○	×	○	○	○	S1(A1)(8/9)
no.7	○	○	○	○	○	○	×	○	○	S1(A1)(8/9)
no.8	○	○	○	○	○	○	○	×	○	S1(A1)(8/9)
no.9	○	○	○	○	○	○	○	○	×	S1(A1)(8/9)

1 診断モデル 1 エージェント

表から, エージェントの故障によって, そのエージェントが持つ情報と診断モデルは失われるが, 残りのエージェントで出来る限りの推論を行っていることが読みとれる. 例えば, 診断モデル S1-A1 を持つエージェント 1 が故障した場合, 残りのエージェントは, 異常原因 S1-X1, S1-X2, S1-X8, S1-X9 を推論する. 従って, 診断モデルが複数のエージェントに渡っていなくてもかなりの頑健性が期待できる.

	a1	a5	a6	a12	a13	a14	a15	a20	a21	結果
no.1	×	○	○	○	○	○	○	○	○	S1(X1),S1(X2),S1(X8),S1(X9)
no.2	○	×	○	○	○	○	○	○	○	S1(A1)(S1(X1) 無し)
no.3	○	○	×	○	○	○	○	○	○	S1(A1)
no.4	○	○	○	×	○	○	○	○	○	S1(A1)(S1(X8) 無し),S1(O7)
no.5	○	○	○	○	×	○	○	○	○	S1(A1)(S1(X9) 無し),S1(O8)
no.6	○	○	○	○	○	×	○	○	○	S1(A1)(S1(O1) 無し)
no.7	○	○	○	○	○	○	×	○	○	S1(A1)(S1(O2) 無し)
no.8	○	○	○	○	○	○	○	×	○	S1(A1)(S1(O3) 無し)
no.9	○	○	○	○	○	○	○	○	×	S1(A1)(S1(O4) 無し)

複数エージェント故障

1 診断モデル 1 エージェントの環境でエージェントを徐々に故障させていく実験を試みた. 表から, 診断情報がいきなり大きく失われる事はなく, 徐々に失われて行く様子が読みとれる. これは, ある 1 エージェントの故障が診断結果に大きな影響を与えることはなく, システムがエージェントの故障に対し頑健性を持つことを示している. 同等の診断能力をもつエージェントを複数定義することによりシステムの頑健性が向上する事を 2 章で述べたが, ここにそのことが実証されたことになる.

	a1	a5	a6	a12	a13	a14	a15	a20	a21	結果
no.1	×	○	○	○	○	○	○	○	○	S1(X1),S1(X2),S1(X8),S1(X9)
no.2	×	×	○	○	○	○	○	○	○	S1(X2),S1(X8),S1(X9)
no.3	×	×	×	○	○	○	○	○	○	S1(X8),S1(X9),S1(O1),S1(O2)
no.4	×	×	×	×	○	○	○	○	○	S1(X9),S1(O1),S1(O2),S1(O7)
no.5	×	×	×	×	×	○	○	○	○	S1(O1),S1(O2),S1(O7),S1(O8)
no.6	×	×	×	×	×	×	○	○	○	S1(O2),S1(O7),S1(O8)
no.7	×	×	×	×	×	×	×	○	○	S1(O7),S1(O8)
no.8	×	×	×	×	×	×	×	×	○	S1(O8)
no.9	×	×	×	×	×	×	×	×	×	無し.

論理回路の複合故障とエージェントの故障数

一度に 2 個以上の論理ゲートが故障する場合についても実験を行ったが. 単一故障時の結果に推論木が増えるだけである. 例えば, S1(A1) と S1(X6) の故障が同時に起こっていたとしても, 診断結果に S1(X6) の木が増えるだけである. 従って実験結果を載せる事はしない.

4.2.3 推論の非単調性

実験結果

図 4.1 の回路で論理ゲート A1 の out ラインに 1-縮退故障が生じたとする. 診断モデルは S1-A1(図 4.8) が用いられる. 実験ではエージェント 13 が誤情報を流したと仮定し, その推

論過程における黒板上の木を出力させてみた. 結果を図 4.7 に示す. なお, この図はプログラムの出力結果から必要な情報を取り出してまとめたものである.

図 4.7 は, step1, step2, .., step6 の順に時間経過している. 注目すべき点は step2-3 と step5-6 の処理である. step2 の状態で故障エージェント 13 が誤情報 “out13=0” を黒板に報告し, 本来正しい推論木である S1-A1 を削除してしまう. しかし, step5 で相互診断の結果が黒板に報告されエージェント 13 が信頼できないことが分かると, その観測情報 “out13=0” は不定情報となり, S1-A1 が削除される理由が無くなる. 結果 step6 で再び S1-A1 が推論されることになる. (非単調性)

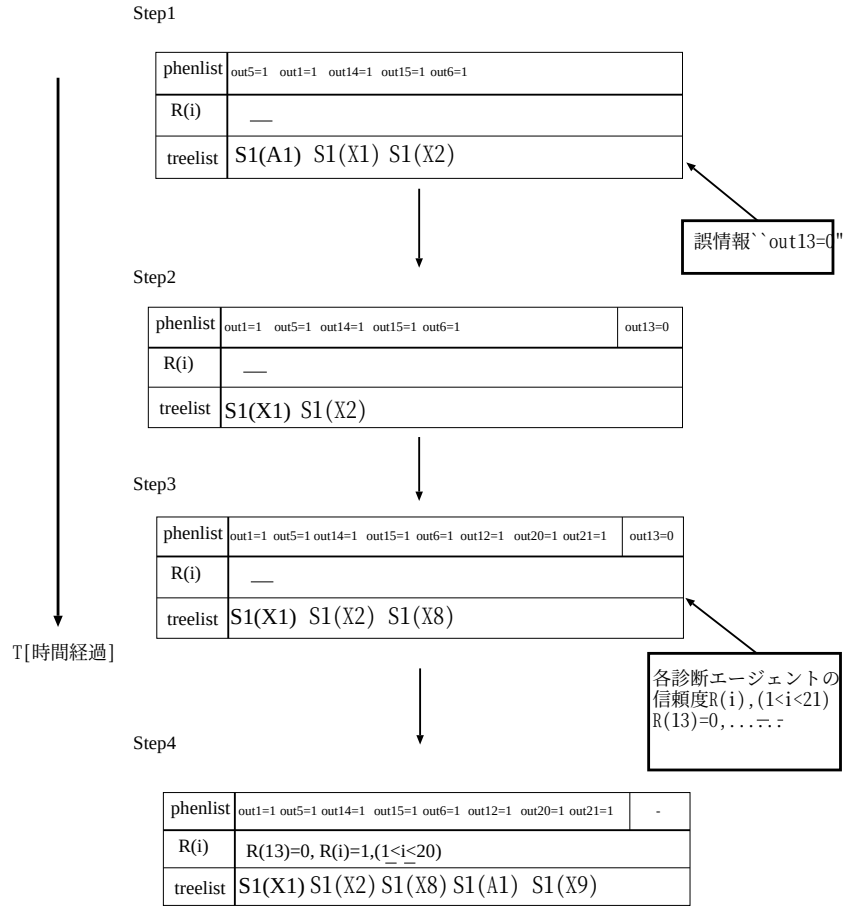

図 4.8: 非単調性 (実験結果)

形式化との比較

この推論過程が, 先に行った非単調推論の形式化に沿うかを調べる. まず準備として以下定義を行う. 尚, 図中 $S1(x)$ は “論理ゲート x が 1-縮退故障している.” ことを示し, $OUT1(x)$ は “論理ゲート x の出力が 1.” であることを示す. これは先の図中の “outA=1”(A はゲート x 担当のエージェント番号) と同義である.

- 診断エージェントが持つ診断モデル.

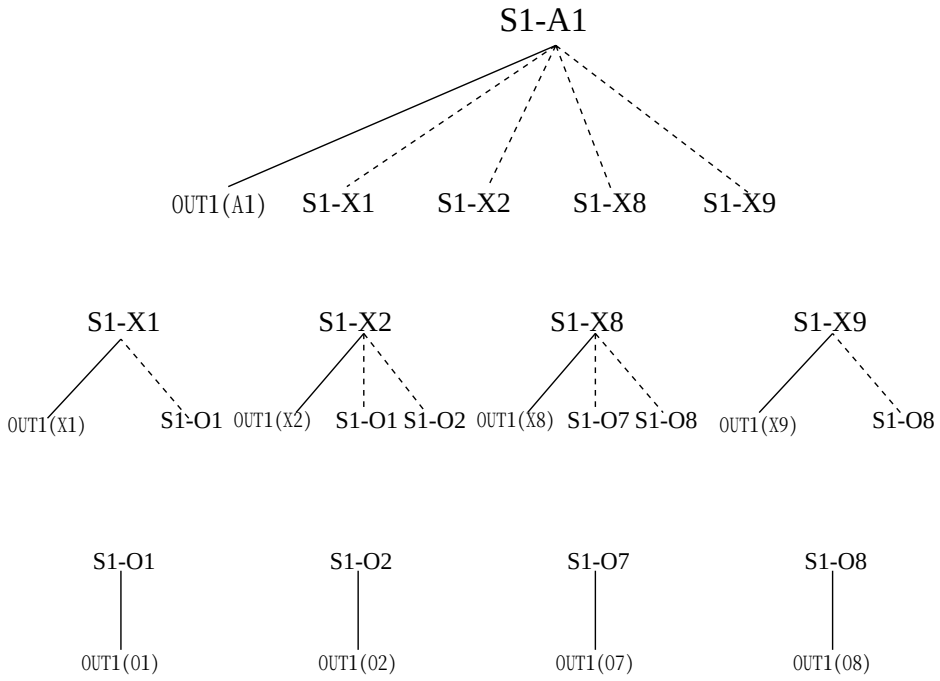

図 4.9: 診断モデル (非単調性実験)

- 各異常原因や中間現象は仮説推論される. 従って, 仮説の否定が報告された場合には, その仮説に基づく現象は推論されてはならない. 以下に異常現象の推論可能条件 (発火条件) を示す.
  - 異常現象 $S1(A1)$.

$$((OUT1(A1) \vee S1(X1) \vee S1(X2) \vee S1(X8) \vee S1(X9)) \wedge$$

$(\neg OUT1(A1) \land \neg S1(X1) \land \neg S1(X2) \land \neg S1(X8) \land \neg S1(X9)) \rightarrow S1(A1)$

- その否定$\neg S1(A1)$.

  $\neg OUT1(A1) \lor \neg S1(X1) \lor \neg S1(X2) \lor \neg S1(X8) \lor \neg S1(X9) \rightarrow \neg S1(A1)$

- 異常現象$S1(X1)$.

  $(OUT1(X1) \lor S1(O1)) \land (\neg OUT(X1) \land \neg S1(O1)) \rightarrow S1(X1)$

- その否定$\neg S1(X1)$.

  $\neg OUT(X1) \lor \neg S1(O1) \rightarrow S1(X1)$

- 異常現象$S1(X2)$.

  $(OUT1(X2) \lor S1(O1) \lor S1(O2)) \land (\neg OUT(X2) \land \neg S1(O1) \land \neg S1(O2)) \rightarrow S1(X2)$

- その否定$\neg S1(X2)$.

  $\neg OUT(X2) \lor \neg S1(O1) \lor S1(O2) \rightarrow S1(X2)$

- 異常現象$S1(X8)$.

  $(OUT1(X8) \lor S1(O7) \lor S1(O8)) \land (\neg OUT(X8) \land \neg S1(O7) \land \neg S1(O8)) \rightarrow S1(X8)$

- その否定$\neg S1(X8)$.

  $\neg OUT(X8) \lor \neg S1(O7) \lor S1(O8) \rightarrow S1(X8)$

- 異常現象$S1(X9)$.

  $(OUT1(X9) \lor S1(O8)) \land (\neg OUT(X9) \land \neg S1(O8)) \rightarrow S1(X9)$

- その否定$\neg S1(X9)$.

  $\neg OUT(X9) \lor \neg S1(O8) \rightarrow S1(X9)$

- 異常現象とその否定$S1(O1), \neg S1(O1)$.

  $OUT1(O1) \rightarrow S1(O1), \neg OUT(O1) \rightarrow \neg S1(O1)$

- 異常現象とその否定$S1(O2), \neg S1(O2)$.

  $OUT1(O2) \rightarrow S1(O2), \neg OUT(O2) \rightarrow \neg S1(O2)$

- 異常現象とその否定$S1(O7), \neg S1(O7)$.

  $OUT1(O7) \rightarrow S1(O7), \neg OUT(O7) \rightarrow \neg S1(O7)$

- 異常現象とその否定$S1(O8), \neg S1(O8)$.

  $OUT1(O8) \rightarrow S1(O8), \neg OUT(O8) \rightarrow \neg S1(O8)$

これら診断モデルとその発火規則を用いて図 4.5 の推論過程 2 の拡張を求める. 定義

$D = \{d : d = d - agt(agt_i) : MR(agt_i) = 1/O(agt_i), (1 \leq i \leq 21)\}$

$E_0 = W = \{x : x = d - agt(agt_i)(1 \leq i \leq 21)\}$

$E_{i+1} = Th(E_i) \cup CGD(D, E_i, E)$

より,

$$E_{step2} = \{S1(A1), S1(X1), S1(X2),$$
$$OUT1(A1), OUT1(X1), OUT1(X2), OUT1(O1), OUT1(O2)\}$$

となる. しかし, ここで誤情報 “OUT0(X9)” が報告されると仮説推論おいて, $\neg S1(X9)$ の発火条件,

$\neg OUT1(X9) \vee \neg S1(O8) \rightarrow \neg S1(X9)$

が満たされ, $\neg S1(X9)$ が発火される. その結果, $\neg S1(A1)$ の発火条件が満たされることになり, 推論結果から S1(A1) が省かれる.

その後 step5 において, 相互診断の結果からエージェント 13 の故障が判明すると観測 “out0(X9)” は無視される. この結果$\neg S1(X9)$ が発火されず, $\neg S1(A1)$ ではなく $S1(A1)$ の発火条件が満たされることになり, 再び “S1(A1)”が推論される. ここに実験結果と理論値が一致したことになる.

4.2.4 考察

診断システムに高い信頼度を持たせる目的で相互認識ネットワークを用い, また, その相互診断の結果を推論に的確に反映させるために非単調推論を導入した. この章の各種実験では, これら二つのプロセスがうまくかみあってシステムの頑健性向上に貢献した事を示している. この章で実証された具体的な事実は以下のようになる.

- 従来の枠組では対応していなかった誤情報に対しての頑健性が得られた. また, 故障エージェント数の増加が推論結果に大きく影響する事はなく, 推論能力は緩やかに低下していくことが確認された.

- エージェントの観測が誤情報であると判明したときに, その事実を適切に推論結果に反映させることが出来た. また, その推論結果推移に非単調性が見られた. また, その推論結果推移過程が Reiter のデフォルト論理と一致することも確認された.

- 並列オブジェクト指向言語 ABCl/f によって並列に記述されたプログラムに, その並列度合に応じた処理の効率化が見られた. 本研究において, 処理の効率化は本質的ではない. 従って, ここでのプログラムの並列度合は診断エージェントの分散度合を意味する.

第 5 章

まとめ

診断システムの研究において, 診断システム自体の頑健性は重要な課題である. しかし, 従来の集中処理方式や分散問題解決では, 大規模化, 分散化されていく診断対象に対して十分な頑健性を保持することは難しい. 分散問題解決はシステムの構成要素が分散しているため, 集中処理方式との比較では頑健性の向上や負荷分散が見られるが, それらは十分でない. 従って, 分散診断を基本とした更なる改良が求められる. その改良法の一つに各エージェントに自己を持たせ, 自己利益を追求させるというマルチエージェントがある. マルチエージェントは以下の点から頑健性に優れた枠組であると言える.

- 自己の利益追求からくる自己防衛機能. 自身に有利な情報以外は無視するなどの利益追求のための判断は, 一般に誤情報は利益をもたらさないことから, 誤情報に惑わされる事なく正しい推論が行える事を示している.
- エージェントの価値の均等化. 各エージェントが自己利益を追求するため, 同等の機能を持つエージェントが定義されることになる. 同等のエージェント複数定義は, その中のエージェントが故障したとしても, システムの機能に大きな影響を与えないことを意味する.

本研究では, このマルチエージェントを診断システム導入し, 分散システムの主な故障に対し頑健性の向上をはかった. また, エージェントの利益追求の判断基準として情報源の信頼度を用いるために, エージェント同士が互いに互いを診断する相互認識ネットワークを, そして, その診断結果を推論に反映させるために非単調推論を各々導入した. その結果得られた成果を以下にまとめる.

- システムの頑健性の向上. 従来の枠組では, 情報の欠落に対してのみ頑健性が報告されていた. それに対し本枠組では誤情報に対しても頑健性を示す事が出来た. また, 各エージェントの故障の影響が推論システム自体及ぶことはなく, その影響は推論の情報量の減少に留まる事を実証した.

- 故障エージェントの誤情報に対応するため, 非単調推論を導入した. あるエージェントに誤情報を流させて, その推論過程を追うことで, 実際に推論結果が非単調に推移している事を確認した. また, この推移は Reiter のデフォルト論理と一致する.

- これら分散診断システムを並列オブジェクト指向言語を用いて, 実際の並列分散環境として実装した. 本稿で述べた実験結果は全て並列分散環境で行ったものである. 並列分散環境によるプログラムの非同期な動きも観測された.

以下に今後の課題をまとめると以下のようになる.

- 黒板導入によるボトルネックの回避. エージェント数が増える程, それに伴うボトルネックが問題になると思われる. 本枠組では, 推論の収束性を考慮して黒板を導入したが, この先はボトルネックの回避が望まれる.

- 仮説機構の充実. 本枠組では仮説推論の仮説選択方法には深く触れていない. 仮説推論は, 各エージェントが仮説を含む部分木を持つことで実現されている. しかし, 診断モデルを部分木として持つことには, メモリの無駄に繋がる等の問題がある. 診断モデルを木として持つ事のメリットと, メモリの無駄になるというデメリットのトレードオフの解決が望まれる.

- モデルをネットワークやプラント診断に適応させること. 本枠組では, 診断対象を論理回路としているが, これは異常現象伝播などのシュミレーションが比較的容易であるという理由に基づいた選択である. したがって, 本来診断対象として想定している, ネットワークやプラントに適応させることが望まれる.

謝辞

本研究を進めるにあたり, 指導教官の東条敏助教授には数多くの御教示を頂きました. また, 國藤研究室 博士後期課程の村川賀彦氏には, 本研究の基本となるアイデア, 及び貴重な助言を頂きました. 佐藤研究室 博士後期課程の小野哲雄氏には適切な助言や示唆を頂きました. そして, 東条研の皆様がたには研究に関する貴重な支援をして頂きましたことを心から感謝致します.

最後に, 常日頃からの両親の支援に心から感謝致します.

参考文献

[1] 村川 賀彦, 診断モデルを持つエージェントの協調による分散診断, 北陸先端科学技術大学院大学修士論文, Feb. 1995

[2] 國藤 進, 鶴巻 宏治, 古川 康一, 仮説選定機構の一実現法, 人工知能学会誌, Vol. 1, No.2, pp.228-237, Dec. 1986

[3] 萩原 兼一, フォールトトレラント分散システム向けアルゴリズム概論, 情報処理, VOL.34, NO.11, pp.1336-1340, Nov. 1993

[4] 上野 晴樹, 小山 照夫, 診断型システム, 知識工学講座 5 エキスパートシステム, オーム社, pp.81-101, 1988

[5] 野口 正一, 三原 幸博, 田村 信介, COM シリーズ, 図解分散処理入門, オーム社, 1987

[6] 木下 哲男, 菅原 研治, エージェント指向コンピューティング, ソフト・リサーチ・センター, 1995

[7] 所 真理雄, マルチエージェントシステム研究の目指すもの, コンピュータソフトウェア, Vol.12, No. 1, pp.78-84, 1995

[8] 大沢 英一, 合理的エージェントによる協同プランスキーマ, コンピュータソフトウェア, Vol.12, No. 1, pp.52-63, 1995

[9] 有馬 淳, 佐藤 健, 非単調推論, 知識情報処理シリーズ 8 知識プログラミング, 共立出版, pp189-214, 1988

[10] Yoav Shoham, Nonmonotonic Logics, Reasoning About Change, The MIT Press, pp.71-94, 1986

[11] 石塚 満, 小林 重信編, 診断型エキスパートシステム, エキスパートシステム, 丸善, pp. 91-121, 1991

[12] 菅原 俊治, 大規模インターネットワーク診断/監視エキスパートシステムについて, 電子情報通信学会論文誌 D-I, Vol.J37-D-I, No.12, pp.990-996, 1990

[13] 松本 一則, 橋本 和夫, 小花 貞夫, 国際電話網の悪化検出/原因推定のための実時間網運用支援エキスパートシステム, 信学技報, SSE93-137, Jan. 1994

[14] Reiter, R. , A logic for default reasoning, Artificial Intelligence 13, pp.81-132, 1980

[15] McDermott, D.,Doyle, J.,Non-monotonic Logic I,Artificial Intelligence 13, pp.41-72, 1980

[16] McDermott, D.,Non-monotonic Logic II:Non-monotonic modal theories, JACM 29, pp.33-57, 1982

[17] McCarthy J.,Circumscription - a form of non-monotonic reasoning, Artificial Intelligence 13, pp.27-39, 1980

[18] 米澤 明憲, 柴山 悦哉, 分散システムのモデル化に向けて, 岩波講座ソフトウェア科学 17, モデルと表現, 岩波書店, pp.167-176, 1992

[19] 米澤 明憲, 柴山 悦哉, 分散システムの並列オブジェクト指向表現, 岩波講座ソフトウェア科学 17, モデルと表現, 岩波書店, pp.177-201, 1992

[20] Smith R., The Contract Net Protocol: Hight Level Communication and Control in a Distributed Problem Solving, IEEE Trans. Comp., Val.29, No.12, pp.1104-1113, 1980

[21] Erman, L.D.,Hayes-Roth, F.,Lesser, V.R.,Reddy, D.R., The Hearsay-II Speech-Understanding System : Integrated Knowledge to Resolve Uncertainty, Comput. Surveys, Vol.12, pp.213-253, 1980

[22] 石田 好輝, Franqis Mizessyn, 免疫型並列分散モデルによるプラントのセンサ自律診断, システム制御情報学会論文誌, Vol.7, No.1 , pp.1-8,1994

[23] 石田 好輝, 免疫型ネットワーク情報モデルと故障診断

[24] 田浦 健次郎, ABCL/f 試作システム操作説明 (暫定版)

付録 A

プログラムソース

付録として, 以下 ABCL/f のソースプログラムをつける. 尚これは, 重要な部分の抜粋で, 具体的には通信用の関数, エージェント内部変数処理メソッド, 黒板内部変数処理メソッド等を挙げる.

A.1 message 関数

```
;;; エージェントの木を更新 ;;;

(defmethod! agent renewal-tree2 (stl trl)
  (declare (st-list stl)(st-list trl)(reply-type unit))
  (let ((inl (get-inlist stl))
        (outl (get-outlist stl))
        (tinl (get-inlist trl))
        (toutl (get-outlist trl))
        (dete detect-tree))
    (dolist (tmp dete)
          (dolist (tm inl)
                (search-hypo-tarbo tmp tm)))
    (dolist (tmp dete)
          (dolist (tm outl)
                (search-hypo-tarbo tmp tm)))
    (dolist (tmp dete)
          (dolist (tm tinl)
                (search-hypo-tarbo tmp tm)))
    (dolist (tmp dete)
          (dolist (tm toutl)
                (search-hypo-tarbo tmp tm)))
    (become unit :detect-tree dete)))

;;; 木が In リスト, Out リストのどちらに入るべきかを決定 ;;;
(defun falt-search (tr)
  (declare (tree tr)(reply-type boolean))
  (let* ((val (get-value tr))
         (vmk (vmark-out val))
         (vme (out-value-name val))
         (ans false))
    (if (= vmk 3) true
      (progn
        (let ((chl (get-child tr)))
          (dolist (tmp chl)
                (setq ans (or (falt-search tmp) ans))))
        ans))
    ))

(defun in-out-select (trl)
  (declare ((list tree) trl)(reply-type (list (list tree))))
  (let ((inl (cast '() (list tree)))
        (outl (cast '() (list tree))))
    (dolist (tmp trl)
          (if (falt-search tmp)
              (setq outl (cons tmp outl))
            (setq inl (cons tmp inl))))
    (list inl outl)))

;;; 黒板に新しい木を書く ;;;

(defun newlef-set (bb tl)
  (declare (blackboard bb)((list tree) tl)(reply-type st-list))
  (let* ((tal (car tl))
         (val (get-value tal))
         (vmk (vmark-out val))
         (dphn (get-dphen bb))
         (bip (get-inlist dphn))
         (bop (get-outlist dphn)))
    (if (= vmk 3)
        (progn
          (when (not (member-t tal bop))
                (setq bop (cons (copy tal) bop))))
      (progn
        (when (not (member-t tal bip))
              (setq bip (cons (copy tal) bip)))))
    (st-list bip bop)))

(defun newtrl-set (tl1 tl2 tl)
  (declare ((list tree) tl1)((list tree) tl2)((list tree) tl)
           (reply-type st-list))
  (dolist (tmp (cdr tl))
          (if (not (falt-search tmp))
              (progn
                (let ((ptl (get-parent tmp)))
                  (when (and (null ptl)(not (member-t tmp tl1)))
                        (setq tl1 (cons (copy tmp) tl1)))))
              (progn
                (let ((ptl (get-parent tmp)))
                  (when (and (null ptl)(not (member-t tmp tl2)))
                        (setq tl2 (cons (copy tmp) tl2)))))))
    (st-list tl1 tl2)))

;;; 黒板上の木を更新 ;;;
```

```
(defun renew-bbtree (bb trl)
  (declare (blackboard bb)((list tree) trl)
           (reply-type (list (list tree))))
  (let* ((tl (get-treel bb))
         (detl (get-inlist tl))
         (notl (get-outlist tl)))
    (dolist (tmp detl)
            (dolist (tm trl)
                    (search-hypo-tarbo tmp tm)))
    (dolist (tmp notl)
            (dolist (tm trl)
                    (search-hypo-tarbo tmp tm)))
    (let* ((tl1 (append detl notl))
           (tll (in-out-select tl1)))
      tll)))

;;;; エージェントによってなされる黒板上の処理のまとめ ;;;;;

(defmethod! blackboard bb-diag2 (agt)
  (declare (agent agt)(reply-type unit))
  (renewal-tree2 agt d-phen treelist)
  (let* ((dete (get-detect-tree agt))
         (atom-io (renew-bbtree self dete))
         (atom-i (car atom-io))
         (atom-o (car (cdr atom-io)))
         (nc (get-nconf agt))
         (anm (get-name nc))
         (phn (newlef-set self dete))
         (atm-io (newtrl-set atom-i atom-o dete)))
    (become unit :treelist atm-io :d-phen phn)))

;;; エージェントの全処理 ;;;
(defun c-diagnosis2 (agt stl)
  (declare (agent agt)((list string) stl)(reply-type unit))
  (dolist (tmp stl)
          (agt-inf agt tmp))
  (let ((sl (agt-test agt)))
    (set-sam agt sl))
  (when (not (null (get-detect-tree agt)))
        (let* ((bb1 (get-bp agt))
               (bb (get-bb bb1)))
          (bb-diag2 bb agt)
          )))

;;; 新たな情報のあと黒板上の木を In, Out に再分割する. ;;;
(defmethod! blackboard reset-bbtree (st rf)
  (declare (string st)(fixnum rf)(reply-type unit))
  (let* ((detl (get-inlist treelist))
         (notl (get-outlist treelist)))
    (dolist (tmp detl)
            (string-search tmp st rf))
    (dolist (tmp notl)
            (string-search tmp st rf))
    (let* ((tl (append detl notl))
           (tll (in-out-select tl))
           (in (car tll))
           (out (car (cdr tll))))
      (become unit :treelist (st-list in out)))))

;;; ポーリング ;;;
(defun new-polling (agt stl)
  (declare (agent agt)((list string) stl)(reply-type unit))
  (let* ((nc (get-nconf agt))
         (bbp (get-bp agt))
         (anm (get-name nc))
         (str (nth 1 (name-lef anm)))
         (ftl (tr-name-list agt str))
         (bb (get-bb bbp))
         (tl (get-treel bb))
         (inl (get-inlist tl))
         (outl (get-outlist tl)))
    (when (or (string= anm "agt14")(string= anm "agt7")
              (string= anm "agt21")(string= anm "agt13")
              (string= anm "agt6")(string= anm "agt20")
              (string= anm "agt1"))
      (printn "========== "anm " start ==========" )
      (printn "********* inlist ********")
      (dolist (tmp inl)
              (display-tree tmp)
              (printn "————"))
      (printn "********* outlist ********")
      (dolist (tmp outl)
              (display-tree tmp)
              (printn "————"))
      (printn "========== end ==========")))
  (dolist (x ftl)
          (dolist (tmp inl)
                  (when (search-hypo tmp x)
                        (progn
                          (dolist (tm stl)
                                  (detect-fault agt tm))
                          (bb-diag2 bb agt)))))
  (dolist (x ftl)
          (dolist (tmp outl)
                  (when (search-hypo tmp x)
                        (progn
                          (dolist (tm stl)
                                  (detect-fault agt tm))
                          (bb-diag2 bb agt)))))
  ))
```

A.2 agent の定義

```
;;; agent のクラス定義 ;;;

(defclass agent ()
  ((list (list tree)) static-tree)
  ((list tree) detect-tree)
  ((list sample) sam)
  (nameconf nconf)
  (point bp)
  (fixnum kind)
  ((list (list fixnum)) know)
  )

;;; agent の各種内部変数を参照するためのメソッド ;;;

(defmethod agent get-static-tree ()
  (declare (reply-type (list (list tree))))
  static-tree)

(defmethod agent get-detect-tree ()
  (declare (reply-type (list tree)))
  detect-tree)

(defmethod agent get-bp ()
  (declare (reply-type point))
  bp)

(defmethod agent get-sam ()
  (declare (reply-type (list sample)))
  sam)

(defmethod agent get-nconf ()
```

```
  (declare (reply-type nameconf))
  nconf)

(defmethod agent get-know ()
  (declare (reply-type (list (list fixnum))))
  know)

(defmethod agent get-kind ()
  (declare (reply-type fixnum))
  kind)

;;; agent の各種内部変数を書き換えるメソッド ;;;

(defmethod! agent set-static-tree (tl)
  (declare ((list (list tree)) tl)(reply-type unit))
  (become unit :static-tree tl))

(defmethod! agent set-detect-tree (tl)
  (declare ((list tree) tl)(reply-type unit))
  (become unit :detect-tree tl))

(defmethod! agent set-bp (tl)
  (declare (point tl)(reply-type unit))
  (become unit :bp tl))

(defmethod! agent set-sam (sl)
  (declare ((list sample) sl)(reply-type unit))
  (become unit :sam sl))

(defmethod! agent set-nconf (nc)
  (declare (nameconf nc)(reply-type unit))
  (become unit :nconf nc))

(defmethod! agent set-know (fll)
  (declare ((list (list fixnum)) fll)(reply-type unit))
  (become unit :know fll))

;;; 診断対象を診断するための関数 ;;;

(defun agt-inf (agt st)
  (declare (agent agt)(string st)(reply-type unit))
  (let* ((know (get-static-tree agt))
         (stl (cast '() (list string))))
    (dolist (tmp know)
            (let* ((tnm (nth 0 tmp))
                   (val (get-value tnm))
                   (vnm (out-value-name val)))
              (setq stl (cons vnm stl))))
    (when (member st stl)
          (dolist (tmp know)
                  (let* ((tnm (nth 0 tmp))
                         (val (get-value tnm))
                         (vnm (out-value-name val))
                         (vnv (vmark-out val)))
                    (when (and (string= vnm st)(not (= vnv 3)))
                          (set-detect-tree agt tmp))
                    )))))

(defun detect-fault (agt st)
  (declare (agent agt)(string st)(reply-type unit))
  (let ((know (get-static-tree agt))
        (stl (cast '() (list string))))
    (dolist (tmp know)
            (let* ((tnm (nth 0 tmp))
                   (val (get-value tnm))
                   (vnm (out-value-name val)))
              (setq stl (cons vnm stl))))
    (when (member st stl)
                (dolist (tmp know)
                        (let* ((tnm (nth 0 tmp))
                               (val (get-value tnm))
                               (nmv (out-value-name val))
                               (vnv (vmark-out val)))
                          (when (and (string= st nmv)(= vnv 3))
                                (set-detect-tree agt tmp))
                          )))))

(defun agt-pro (agt stl)
  (declare (agent agt)((list string) stl)(reply-type unit))
  (dolist (tmp stl)
          (agt-inf agt tmp)
          (detect-fault agt tmp)))

;;; 相互認識ネット用の関数 ;;;

(defun test (agt)
  (declare (agent agt)(reply-type (list fixnum)))
  (let ((kl (get-know agt))
        (knd (get-kind agt))
        (fl (cast '() (list fixnum))))
    (dolist (tmp kl)
            (let ((in1 (nth 0 tmp))
                  (in2 (nth 1 tmp))
                  (out (nth 2 tmp)))
              (cond ((= knd 1)
                     (if (and (= in1 1)(= in2 1))
                         (if (= out 1)
                             (setq fl (cons 1 fl))
                             (setq fl (cons -1 fl)))
                         (if (= out 0)
                             (setq fl (cons 1 fl))
                             (setq fl (cons -1 fl)))))
                    ((= knd 2)
                     (if (and (= in1 0)(= in2 0))
                         (if (= out 0)
                             (setq fl (cons 1 fl))
                             (setq fl (cons -1 fl)))
                         (if (= out 1)
                             (setq fl (cons 1 fl))
                             (setq fl (cons -1 fl)))))
                    ((= knd 3)
                     (if (and (= in1 in2))
                         (if (= out 0)
                             (setq fl (cons 1 fl))
                             (setq fl (cons -1 fl)))
                         (if (= out 1)
                             (setq fl (cons 1 fl))
                             (setq fl (cons -1 fl)))))
                    (true unit))))
    fl))

(defun test-a (agt)
  (declare (agent agt)(reply-type boolean))
  (let ((fl (test agt)))
    (member-f -1 fl)))

(defmethod agent agt-test ()
  (declare (reply-type (list sample)))
  (let ((sl (cast '() (list sample))))
    (dolist (tmp sam)
            (let ((tes (get-test tmp))
                  (agt (get-dagt tmp)))
              (if (test-a agt)
                  (setq sl (cons (sample -1 agt) sl))
                  (setq sl (cons (sample 1 agt) sl)))))
    sl))
```

A.3 blackboard 定義

```
;;; クラス定義 ;;;

(defclass blackboard ()
  (st-list d-phen)
  (st-list treelist)
  (fixnum hist))

;;; blackboard の内部変数参照用メソッド ;;;

(defmethod blackboard get-dphen()
  (declare (reply-type st-list))
  d-phen)

(defmethod blackboard get-treel()
  (declare (reply-type st-list))
  treelist)

(defmethod blackboard get-hist()
  (declare (reply-type fixnum))
  hist)

;;; blackboard の内部変数書き換え用メソッド ;;;

(defmethod! blackboard set-dphen (s)
  (declare (st-list s)(reply-type unit))
  (become unit :d-phen s))

(defmethod! blackboard set-markb (t)
  (declare (tree t)(reply-type unit))
  (let ((inl (get-inlist d-phen))
        (outl (get-outlist d-phen))
        (nl (cast '() (list tree))))
    (setq nl (cons t inl))
    (become unit :d-phen (st-list nl outl))))
(defmethod! blackboard set-treel (s)
  (declare (st-list s)(reply-type unit))
  (become unit :treelist s))

(defmethod! blackboard inc-hist ()
  (declare (reply-type unit))
  (become unit :hist (+ 1 hist)))
```

A.4 診断モデル

```
;;; defclass tree ;;;
(defclass tree ()
  ((list tree) parent)
  (node-value value)
  ((list string) e-subtree)
  ((list tree) child)
)

(defmethod tree get-value ()
  (declare (reply-type node-value))
  value)

(defmethod tree get-child ()
  (declare (reply-type (list tree)))
  child)

(defmethod tree get-parent ()
  (declare (reply-type (list tree)))
  parent)

(defmethod tree get-e-subtree ()
  (declare (reply-type (list string)))
  e-subtree)

(defmethod! tree set-parent (tr)
  (declare ((list tree) tr)(reply-type unit))
  (become unit :parent tr))

(defmethod! tree set-child (cl)
  (declare ((list tree) cl)(reply-type unit))
  (become unit :child cl))

(defmethod! tree set-sub (cl)
  (declare ((list string) cl)(reply-type unit))
  (become unit :e-subtree cl))

(defmethod! tree detect (vl)
  (declare (node-value vl)(reply-type unit))
  (become unit :value vl))

(defmethod! tree init-node-value (cl)
  (declare ((list tree) cl)(reply-type unit))
  (if (not (null child))
      (progn
        (let ((co (cast '() real)))
          (dolist (tmp cl)
                  (setq co (+ 1.0 co)))
          (become unit :value (set-node-value value (/ 1.0 co)))))
    (become unit :value (set-leaf-value value 0))))

(defmethod tree copy ()
  (declare (reply-type tree))
  (let ((newchild (cast '() (list tree)))
        (newsub (cast '() (list string))))
    (dolist (tmp child)
            (let ((nch (copy tmp)))
              (setq newchild (cons nch newchild))))
    (dolist (tmp e-subtree)
            (setq newsub (cons tmp newsub)))
    (let ((tr (tree (cast '() (list tree)) value
    (reverse newsub) (reverse newchild))))
      (dolist (tmp newchild)
              (let ((pt (get-parent tmp)))
                (set-parent tmp (cons tr pt))))
      tr)))

(defun copy-r (tr)
  (declare (tree tr)(reply-type tree))
  (let ((newpt (cast '() (list tree)))
        (pt (get-parent tr)))
    (dolist (tmp pt)
            (setq newpt (cons (copy-r tmp) newpt)))
    (let ((ntr (copy tr)))
      (set-parent ntr newpt)
      ntr)))

;;; *search leaf ;;;
(defun search-leaf (tr sr)
  (declare (tree tr)(string sr)(reply-type boolean))
  (let* ((cl (get-child tr))
         (vl (get-value tr))
         (ns (out-value-name vl))
         (nv (vmark-out vl)))
    (if (null cl)
        (if (string= ns sr)
            (progn (detect tr (value-mark-leaf vl))true)false)
```

```
    (let ((true-false false))
      (dolist (tmp cl true-false) :reply-type boolean
              (when (search-leaf tmp sr)(return true)))))))



(defun search (tr sr)
  (declare (tree tr)(string sr)(reply-type boolean))
  (let* ((cl (get-child tr))
         (vl (get-value tr))
         (ns (out-value-name vl))
         (nv (vmark-out vl)))
    (if (null cl)
        (if (string= ns sr) true false)
      (let ((true-false false))
        (dolist (tmp cl true-false) :reply-type boolean
                (when (search tmp sr)(return true)))))))

(defun search-hypo (tr sr)
  (declare (tree tr)(string sr)(reply-type boolean))
  (let* ((cl (get-child tr))
         (est (get-e-subtree tr))
         (true-false false))
    (if (member sr est) true
      (dolist (tmp cl true-false) :reply-type boolean
              (setq true-false
                    (or true-false (search-hypo tmp sr))))
      )))

;;; for leaf-del ;;;
(defun string-search (tr st rf)
  (declare (tree tr)(string st)(fixnum rf)(reply-type unit))
  (let* ((cl (get-child tr))
         (vl (get-value tr))
         (ns (out-value-name vl))
         (nv (vmark-out vl)))
    (if (null cl)
        (progn
          (when (string= ns st)
                (let ((nvl (setvalue vl (* nv rf))))
                  (detect tr nvl))))
      (dolist (tmp cl)
              (string-search tmp st rf)))))

;;; for joint sub-tree to parent-tree ;;;
(defun tree-chang (ch n chl)
  (declare (tree ch)(tree n)((list tree) chl)(reply-type (list tree)))
  (let ((l (cast '() (list tree)))
        (st (out-value-name (get-value ch)))
        (nt (out-value-name (get-value n))))
    (dolist (tmp chl)
            (let* ((a (out-value-name (get-value tmp))))
              (if (string= nt a)
                  (setq l (cons ch l))
                (setq l (cons tmp l)))))
    (reverse l)))



(defun tree-chang-s (ch n chl)
  (declare (tree ch)(string n)((list tree) chl)(reply-type (list tree)))
  (let ((l (cast '() (list tree)))
        (st (out-value-name (get-value ch))))
    (dolist (tmp chl)
            (let* ((a (out-value-name (get-value tmp))))
              (if (string= n a)
                  (setq l (cons ch l))
                (setq l (cons tmp l)))))
    (reverse l)))

(defun del-string (st stl)
  (declare (string st)((list string) stl)(reply-type (list string)))
  (let ((l (cast '() (list string))))
    (dolist (tmp stl)
            (if (string= st tmp)
                (setq l l)
              (setq l (cons tmp l))))
    l))

(defun set-subtree (rt st)
  (declare (tree rt)(tree st)(reply-type unit))
  (let* ((mn (get-e-subtree rt))
         (n (get-value st))
         (pl (get-parent st))
         (nm (out-value-name n)))
    (dolist (x mn)
            (let ((c (get-child rt)))
              (when (string= nm x)
                    (progn
                      (let ((nnm (del-string x mn)))
                        (set-parent st (cons rt pl))
                        (set-child rt (cons st c))
                        (set-sub rt nnm)))
                    )))
    ))

(defun search-hypo-tarbo (tr str)
  (declare (tree tr)(tree str)(reply-type unit))
  (let* ((cl (get-child tr))
         (est (get-e-subtree tr))
         (val (get-value str))
         (tnm (out-value-name val)))
    (if (member tnm est)
        (set-subtree tr str)
      (dolist (tmp cl)
              (search-hypo-tarbo tmp str)))))

;;; display tree ;;;
(defun display-tree (tr)
  (declare (tree tr)(reply-type unit))
  (let ((root (get-value tr))
        (child (get-child tr)))
    (print "(")
    (if (not (null child))
        (progn
          (print "[" (out-value-name root) "
          "(out-value-confident root)" "(out-value-rate root) "]" )
          (printn)
          (dolist (tmp child)
                  (print "    ")
                  (display-tree tmp)))
      (print "[" (out-value-name root) " " (vmark-out root) "]"))
    (printn ")")
    )
  )

;;; search ;;;
(defun search-leaf-unit (tr sr)
  (declare (tree tr)(string sr)(reply-type unit))
  (let* ((cl (get-child tr))
         (vl (get-value tr))
         (ns (out-value-name vl))
         (nv (vmark-out vl)))
    (if (null cl)
        (when (string= ns sr)
              (detect tr (value-mark-leaf vl)))
      (dolist (tmp cl)
        (search-leaf-unit tmp sr)))))
```

付録 B

実行結果

agent1, ゲート A1 故障

```
************* inlist *************
([S1(X9) 0 1]
    ([S1(O8) 0 1]
    ([out21=1 1])
)
    ([out13=1 1])
)
———————-
([S1(X1) 0 1]
    ([S1(O1) 0 1]
    ([out14=1 1])
)
    ([out5=1 1])
)
———————-
([S1(X2) 0 1]
    ([S1(O2) 0 1]
    ([out15=1 1])
)
    ([S1(O1) 0 1]
    ([out14=1 1])
)
    ([out6=1 1])
)
———————-
([S1(X8) 0 1]
    ([S1(O8) 0 1]
    ([out21=1 1])
)
    ([S1(O7) 0 1]
    ([out20=1 1])
)
    ([out12=1 1])
)
———————-
```

agent1, ゲート A2 故障

```
************* inlist *************
([S1(A2) 0 1]
    ([S1(X6) 0 1]
    ([S1(O6) 0 1]
    ([out19=1 1])
)
    ([S1(O5) 0 1]
    ([out18=1 1])
)
    ([out10=1 1])
)
    ([S1(X3) 0 1]
    ([S1(O3) 0 1]
    ([out16=1 1])
)
    ([S1(O2) 0 1]
    ([out15=1 1])
)
    ([out7=1 1])
)
    ([S1(X2) 0 1]
    ([S1(O2) 0 1]
    ([out15=1 1])
)
    ([S1(O1) 0 1]
    ([out14=1 1])
)
    ([out6=1 1])
)
    ([out2=1 1])
)
———————-
```

agent1, ゲート A3 故障

```
************* inlist *************
([S1(A3) 0 1]
    ([S1(X7) 0 1]
    ([S1(O7) 0 1]
    ([out20=1 1])
)
    ([S1(O6) 0 1]
    ([out19=1 1])
)
    ([out11=1 1])
)
    ([S1(X5) 0 1]
    ([S1(O5) 0 1]
    ([out18=1 1])
)
    ([S1(O4) 0 1]
    ([out17=1 1])
)
    ([out9=1 1])
```

```
)
    ([S1(X4) 0 1]
    ([S1(O4) 0 1]
    ([out17=1 1])
)
    ([S1(O3) 0 1]
    ([out16=1 1])
)
    ([out8=1 1])
)
    ([S1(X3) 0 1]
    ([S1(O3) 0 1]
    ([out16=1 1])
)
    ([S1(O2) 0 1]
    ([out15=1 1])
)
    ([out7=1 1])
)
    ([out3=1 1])
)
————-
```

```
)
    ([out9=1 1])
)
    ([S1(X4) 0 1]
    ([S1(O4) 0 1]
    ([out17=1 1])
)
    ([S1(O3) 0 1]
    ([out16=1 1])
)
    ([out8=1 1])
)
    ([out4=1 1])
)
————-
```

agent1, ゲート A4 故障

```
********** inlist **********
([S1(A4) 0 1]
    ([S1(X9) 0 1]
    ([S1(O8) 0 1]
    ([out21=1 1])
)
    ([out13=1 1])
)
    ([S1(X8) 0 1]
    ([S1(O8) 0 1]
    ([out21=1 1])
)
    ([S1(O7) 0 1]
    ([out20=1 1])
)
    ([out12=1 1])
)
    ([S1(X7) 0 1]
    ([S1(O7) 0 1]
    ([out20=1 1])
)
    ([S1(O6) 0 1]
    ([out19=1 1])
)
    ([out11=1 1])
)
    ([S1(X6) 0 1]
    ([S1(O6) 0 1]
    ([out19=1 1])
)
    ([S1(O5) 0 1]
    ([out18=1 1])
)
    ([out10=1 1])
)
    ([S1(X5) 0 1]
    ([S1(O5) 0 1]
    ([out18=1 1])
)
    ([S1(O4) 0 1]
    ([out17=1 1])
```