第 6 章

適用例 1: 電力制御用ソフトウェア

6.1 緒言

本章では、VOGUE の実際のソフトウェアの 3 次元視覚化への適用例として、電力制御用ソフトウェアのモジュール構造の視覚化、及びそれに時間軸を導入して発展させたプロセス・モニタを取り上げる。前者はモジュール階層構造を 3 次元領域図として表現することで、モジュール階層と静的タスク間通信を同時に表示するものである。このタスク間通信は起動方式によって階層的に管理でき、VOGUE の図形数制御機能のうちオブジェクト指向 DB に基づいた手法が示される。また、このグラフ構造における Fractal Views によるノード数制御機能が示される。後者は東京電力における実機を稼働させることで得られたプロセス実行状態のトレースを 3 次元視覚化するものであり、障害発生時と平常時とのシステムの挙動の相違を視覚的パターンとして表現する。

実際の電力制御システム全体は、複数台計算機がメッセージ送信方式によって通信を行ないながら協調動作する並列分散システムであるが、取得できるトレースデータは 1 台の計算機における実行履歴であり、現段階では各計算機毎に得られたトレースデータから、計算機間メッセージ通信を再現することは困難である。そこで、将来的に各計算機間のメッセージ送信の履歴が記録できるようになった場合の 3 次元視覚化の有効性を示すために、小規模ながらメッセージ伝達によって協調作業を行なう並列分散システムとしての本質を持つ分散マニピュレータ [51] を例にとり、通常状態と事故時の各計算機間におけるメッセージ伝達の視覚化を行ない、これを用いて並列分散システムのデバッグにおける 3 次元視覚化の有効性に関する議論を行なう。ただし、

いづれの例においてもデータは意図的に作成されたものではなく、実際のシステムを稼働することで得られた点が特徴である。

以下、6.2 節において対象とした電力制御用ソフトウェアについて解説し、6.3 節においてモジュール構造視覚化の実現方式について述べ、続く 6.4 節で図形削減手法の適用を行なう。さらに 6.5 節において時間軸を導入したプロセス・モニタへとツールを拡張し、トレース情報の視覚化を行なう。6.6 節では作成したプロセス・モニタを用いて典型的障害事例の視覚化を行ない、視覚化の意義、3 次元化の意義について述べる。さらに 6.7 節では分散マニピュレータのトレースの視覚化を行ない、最後に 6.8 節で本章のまとめを述べる。

6.2 電力制御用ソフトウェアとタスク間通信

6.2.1 電力制御システムの概略

電力制御システムは巨大システムの典型例であり、その制御用ソフトウェアは Programming in the Large の典型例である。東京電力では現在電力系統制御の自動化が進められているが、計算機技術とネットワーク技術の発達により高度な機能が実現可能となった。その例は地方配電系に導入した設備総合自動化システムである。この自動化システムでは、電力系統や機器の監視、記録統計、平常時あるいは事故時の機器操作等、電力系統の運用上必要となる様々な機能が自動化されている。一方で、自動化システムの適用範囲と処理能力の拡大は、大規模システムの設計・製作の困難さを増加させているのが現状である。

一般的にソフトウェア構築には、要求仕様記述、ソフトウェア・プログラム設計、動作テストの各段階が踏まれるが、こうした大規模分散システムの開発においては、計算機同士あるいはタスク同士の協調動作等、システム全体の挙動の記述・検証が必要となり、視覚化の重要性が認識されている。

電力制御システムは、変電所システム、給電所システム、総合制御所システムという大きな 3 つのシステムに分類され、各システムでは各々 1 台の計算機[1]が稼働し、システム同士相互に通信を行なっている。各システム内の処理は幾つかのサブシステムに分類される。例として総合制御所システム内には、監視サブシステム、操作サブシステム、マン・マシン・インタフェース・サブシステム、情報送受信サブシステム等がある。また、各サブシステムの構成要素はタスクと呼ばれる。各タスクはソースレベルで幾つかのモジュールに分割されており、例えば FORTRAN で記述されたモジュールは、幾つものサブルーチンやファンクションから構成されている。つまり全体として、制御用ソフトウェアは Figure 6.1のような階層的構造をなしている。

この電力制御システム全体を対象とするとあまりに規模が大きくなりすぎるため、本論文で対象とするのは、総合制御所システムとそのタスク間通信のレベルまでに限定する。

[1]実際には各々もう 1 台の計算機からなる副システムが、異常発生時に備え待機している

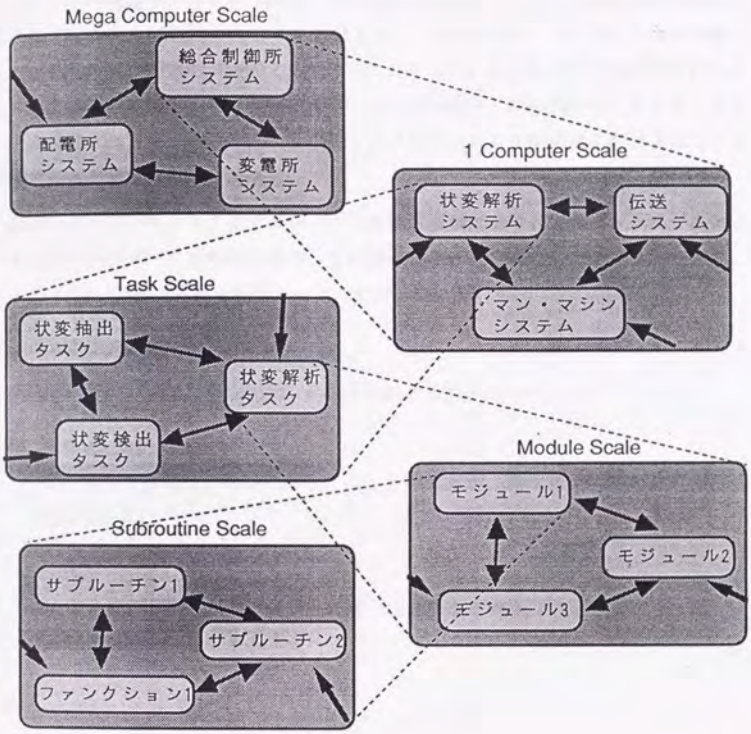


Figure 6.1: 電力制御用ソフトウェアのモジュール構造

6.2.2 電力制御システムにおけるタスク間通信

システムの設計・運用にあたり基本的単位となるのは前述したタスクであり、タスクとタスク間通信を中心として開発・保守が行なわれる。Figure 6.2は現在開発に利用されている総合制御所システムにおけるタスク間関連図で、67 個のタスクが箱として、またタスク間通信が矢印として表現されている。ただし比較的重要度の低いタスクや 2 次元的に記述するのが困難なメッセージ伝達は、図の判りやすさを損なわないため省略されているが、本来ならば全タスク及び全タスク間通信が記述されることが望ましいと考えられる。

前述のように、各タスクはサブシステムの構成要素であり、各々メッセージ送信という方式で他のタスクを起動するが、その際起動ルーチンには以下の 3 種類があり、各々が Table 6.1に示される起動ルーチンを持っている。

1. リアルタイム・システム・サービス
   他プロセスを QUEUE 起動、PASS 起動するためのルーチン

2. コモン・サブルーチン
   他プロセスを起動する事に関しては 1 と同じだが、副システム用の処理が追加されている。

3. 共通ライブラリ
   1、2 と機能は同じだが、エラー処理が追加されている。

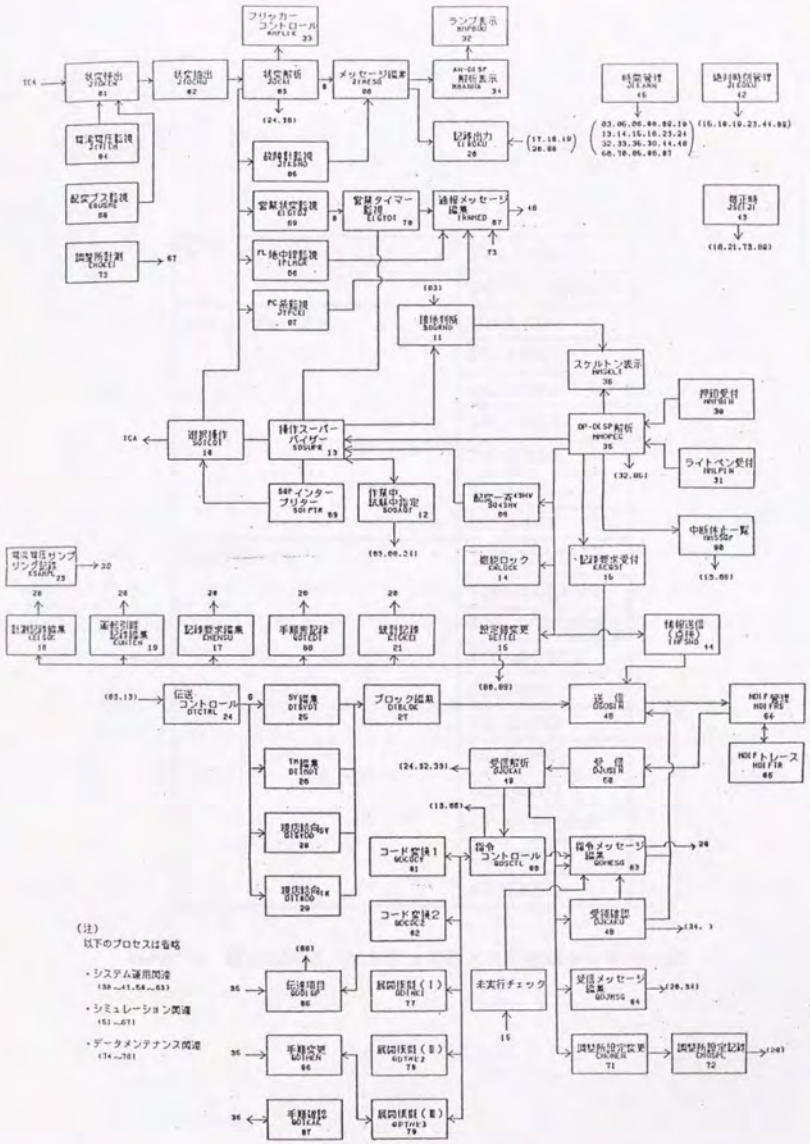


Figure 6.2: タスク間関連図

リアルタイム・システム・サービス	SYS_PASS
	RQS_QUEENQ
コモンサブルーチン	DU_UCEQ
	DU_ENQU
	DU_UCPS
	DU_ENQ2
	DU_TAEQ
	ALMSPS
	DU_TAPS
共通ライブラリ	TNS_PASS
	TNS_QUEENQ
	TNS_SCHDWKE
	TNS_QUEUE
	CL_TAPS
	CL_UCEQ
	CL_DCPS
	CL_DENQ
	CL_UCEQNW
	CL_MSPS
	CL_TAEQ

Table 6.1: 電力制御用ソフトウェアのタスク起動ルーチン一覧

6.3 VOGUE によるタスク間通信の視覚化

本節では、次節で述べるプロセス・モニタのための準備として、また VOGUE の持つ図形削減機能の例題として、前節で解説した総合制御所サブシステムを例にとり、そのタスク間通信とモジュール構造の視覚化を行なう。ここで実現する 3 次元視覚化は第 3 章で述べた 3 次元の " 弱い " 利用法の 1 つであり、十分 2 次元ツールでも対応可能であるが、タスク間関連図では書ききれなかったタスク及びタスク間通信をも表示し、かつ着目するタスクに応じて表示されるタスクを削減しながら表示することができる。

6.3.1 対象のモデル化

システム、サブシステム、タスクは各々クラス system、クラス subsystem、クラス task として定義し、個々のタスク等はこれらのインスタンスとして表現される。また、システム、各サブシステム、各タスクは名前を持ち、これは name スロットが保持することとする。各インスタンス間の関係としては、第 1 にシステムとサブシステム間、サブシステムとタスク間の包含関係を表す partof-link を定義する。つまり作成される全インスタンスは総合制御所をルートとした、partof-link による木構造として表現される。ここまでのモデル定義とクラス階層を Figure 6.3に示す。

次に、前節で説明したタスク間通信の関係に基づき、タスク間通信については、Table 6.1の分類を階層的に表現することを考える。具体的には、まずタスク間通信全てを抽象的に表現するクラスを message-link として定義し、次に表のうちリアルタイム・システム・サービス、コモンサブルーチン、共通ライブラリを各々 rss-link、cs-link、cl-link として message-link のサブクラスとして定義する。さらに具体的な各起動ルーチンをこれらのサブクラスとして定義する。つまり全体として Figure 6.4のようなクラス階層で表現することにする。

```
(define-class system ()
  ((name :initarg :name :accessor name)) )
(define-class subsystem ()
  ((name :initarg :name :accessor name)) )
(define-class task ()
  ((name :initarg :name :accessor name)) )

(define-link partof-link () ())
```

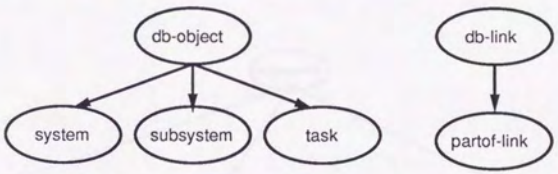


Figure 6.3: モジュール構造のモデル化とデータベース内のクラス階層

```
(define-link message-link () ())

(define-link rss-link (message-link) ())
(define-link sys-pass (rss-link) ())
(define-link rqs-queenq (rss-link) ())

(define-link cs-link (message-link) ())
(define-link du_uceq (cs-link) ())
(define-link du-enqu (cs-link) ())
(define-link du-ucps (cs-link) ())
(define-link du-enq2 (cs-link) ())
(define-link du-taeq (cs-link) ())
(define-link almsps (cs-link) ())
(define-link du-taps (cs-link) ())

(define-link cl-link (message-link) ())
(define-link tns-pass (cl-link) ())
(define-link tns-queenq (cl-link) ())
(define-link tns-schdwke (cl-link) ())
(define-link tns-queue (cl-link) ())
(define-link cl-taps (cl-link) ())
(define-link cl-uceq (cl-link) ())
(define-link cl-dcps (cl-link) ())
(define-link cl-denq (cl-link) ())
(define-link cl-uceqnw (cl-link) ())
(define-link cl-msps (cl-link) ())
(define-link cl-taeq (cl-link) ())
```

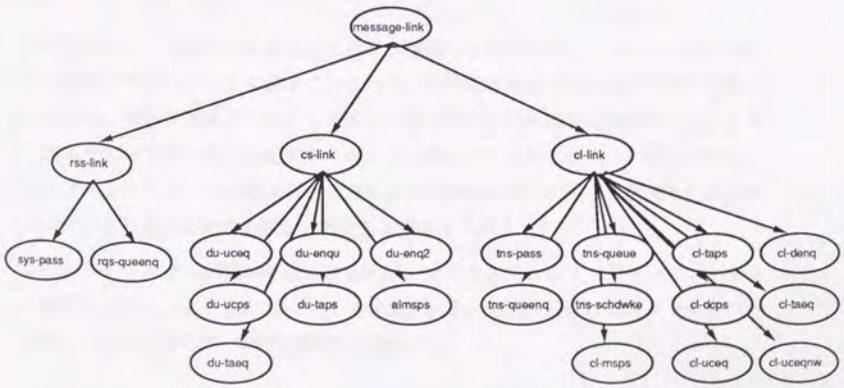


Figure 6.4: タスク間通信のモデル化とデータベース内のクラス階層

6.3.2 グラフへの描画

システム、サブシステム、タスクのインスタンスは、グラフ上で各々ノードとして表示するが、木構造として表現すると階層関係の把握は可能な一方、終端ノードとして表現されるタスク間関連は表現することが不可能となる。 よって、各階層間の階層関係はスケール変換に基づいた 3 次元領域図として表現する。スケール変換は指定されたフラクタル次元に基づくものとし、本例の場合 1.3 とする。タスク間通信はクラス毎に異なる色を割り当てることで識別性を高めるが、色の決め方に必然性はない。色、ノードの形が持つ意味を論じることは本論文の範囲外である。また partof-link の色は無色にする。

Figure 6.5がこうして実現されたモジュール構造の視覚化である。一番外側の大きな立方体が総合制御所システムを、その中にある中程度の立方体がサブシステムを、その中に含まれる立方体が各タスクを表現するノードである。階層の深さによりノードの透明度を変えることにより、タスク間通信を見ながらもその上のサブシステム間の関連も把握できる点が特徴である。各タスク間通信はクラスごとに色の異なるリンクとして表現され全部で 286 個ある。ノード・リンクの重なりは図形を回転させることで回避でき、ステレオ・モードあるいはポリマス・モードにすることで 3 次元立体視が可能である。ただしポリマス・モードでは既にグラフィックスの反応性が悪くなっている。

表示例ではノードのラベルを消してある。現在の 3 次元グラフィックス上では表示する文字列をポリゴンとして扱っているため、不用意に表示すると図の理解を困難なものとする。従来 3 次元グラフィックス上での文字の利用に対する需要がなかったため、読みやすい文字の表示に関するサポートが遅れていることが 1 つの原因である。また、アイコンとラベルの組み合わせ方による認識度の違いに関する研究も行なわれているが、さらなる研究の必要な分野であると考えられる。

特定のサブシステムの詳細を見るためには、サブシステムのノードをマウスクリックし画面左上のメニューからズームインを選択する。図形の回転にはスクロールバーを使用し、 xyz 各軸方向への平行移動も行なえる。

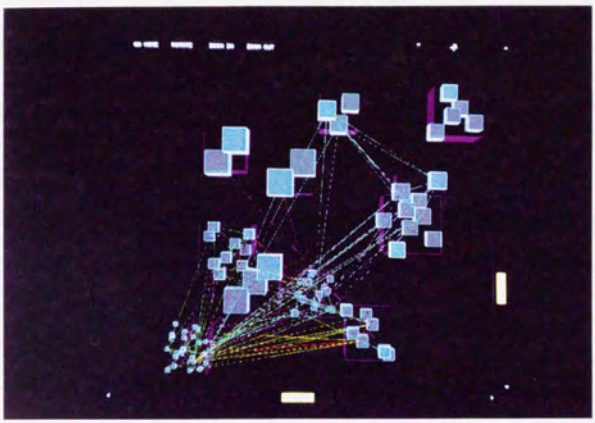


Figure 6.5: VOGUE による電力制御用ソフトウェアのモジュール構造の視覚化

6.4 表示情報量の制御

3 次元に実現されたタスク間通信図は、サブシステム間の関連を漠然と理解することができ、またリンク同士の交差も視点の変更によって回避することができるが、表示されているノード・リンク数が多過ぎて、あるノードあるいはリンクに着目した時、それに関連するノードを識別するのは困難である。VOGUE では 3 種類の図形削減法で図の簡単化を行なった。

6.4.1 タスク起動方式による情報量の制御

タスク間通信はその起動方式によりデータベースによって階層的に管理されているため、注目すべきクラスあるいはリンクが、明らかに定まっている場合には、データの階層性に基づく情報削減手法が有効であると考えられる。

例えば Figure 6.5は初期状態で全タスク通信が見えている状態である。つまり Figure 6.4において全てのリンクが focus されている状態である。次に、リアルタイム・システム・サービスへと着目点を特殊化するには

```
> (focus-link g 'rss-link)
```

を実行する。これにより、rss-link とそのサブクラス以外のリンクは消去される。つまり Figure 6.4の関係から、rqs-queenq と sys-pass だけが表示され (Figure 6.6)、表示されているリンクの数は 91 個に削減される。さらに rqs-queenq へと着目点を特殊化するには、

```
> (focus-link g 'rqs-queenq)
```

を実行する。結果として rqs-queenq だけが表示され (Figure 6.7)、表示リンク数は 66 個に削減される。

本例ではリンクに対する着目点の特殊化を示したが、モデル化の段階でクラスを階層的に作成しておける場合には、ノードに対してもこのような処理が可能である。また Figure 6.4のモデルを変更することでより細かい制御が可能となる。例えば、Table 6.1はタスク起動方式から 3 種類に分類しているが、それぞれのカテゴリにおいて類似した名前のルーチン (例えば、rqs-queenq と tns-queenq) は同じ性質を持ってい

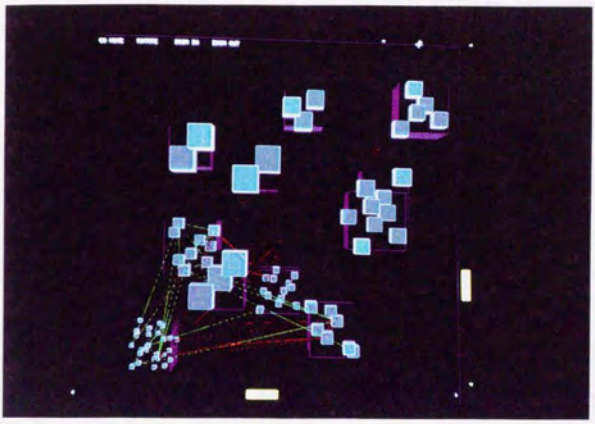


Figure 6.6: RSS-LINK へのフォーカス

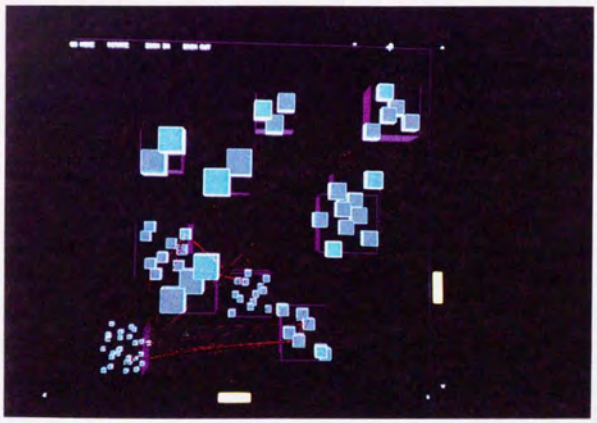


Figure 6.7: RQS-QUEENQ へのフォーカス

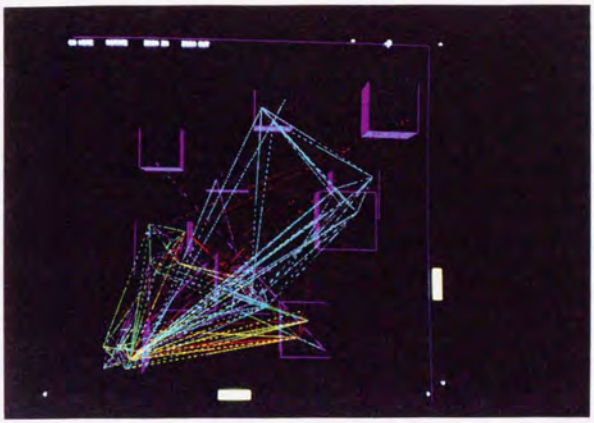


Figure 6.8: 木の深さに基づく図形数制御

る。CLOS を拡張して作成したデータベースは多重継承をサポートするため、これらの関係を多重継承の形でモデル化することが可能となる。こうした多重継承を利用したモデル化により、さらに詳細な情報制御が行なえる。

6.4.2 木の深さによる情報量の制御

次に、着目点からの論理的距離に基づいた図形数制御手法について述べる。Figure 6.8は、総合制御所に着目した状態で距離 1 のノードまで表示した例である。結果として、全タスクを消去したサブシステム間の関連図が得られる。また、距離を 2 とすることで全タスクが表示される。さらに任意のタスクに着目した場合には、そのタスクが属しているサブシステム、及びそのタスクと直接通信を行なっているタスクが表示されるが、何度も述べるように問題点としては、木の分岐数によって表示されるノード数の差が大きいことが挙げられる。

この手法は、着目するノードに対して直接関係を持ったノードを調べる場合等に有効な手法であると考えられる。

6.4.3 Fractal Views による情報量の制御

次に、着目するクラスが明らかでない場合の図形数制御手法として Fractal Views の適用を試みた。Figure 6.9から Figure 6.12はあるタスクに着目した時、Fractal View によっていかにノード数が削減されるか示した例である。 具体的には、画面右下の中立方体のほぼ中央にあるタスクを選択した状態で、次元を 2.5 に固定し、閾値を次第に大きくしていった時の様子である。関係の少ない部分から消えていくのが見て取れる。

Fractal Views の情報量制御能力は第 4 章で数値的に検証したが、ここで再び実際のタスク間関連を例としてその有効性を以下のような実験で検証した。

図のタスク間関連図においてフラクタル次元 FD を一定に保った状態で、任意のノードを着目点として選択する。閾値 k を 0.05 から 0.50 まで離散的に変化させ、グラフ全体で表示されるノード数を記録するという操作をグラフ上の全ノードに対して行なう。この一連のプロセスをフラクタル次元 FD を 1.0 から 3.5 まで離散的に変化させた結果を記録した。Figure 6.13から Figure 6.18のグラフは任意の k について表示されたノード数が最大の場合、最小の場合、そして全ノードでの平均値を求め、横軸を k、縦軸を表示されたノード数としてフラクタル次元別にプロットしたものである。閾値に従って表示されるノード数の帯域が滑らかに変化していくのがわかる。

表示されるノード数をさらに詳細に分析するために $FD = 2.5$ の場合について、各々の k により着目ノードごとに表示される全ノード数を調べた。Figure 6.19から Figure 6.22は、同一次元、同一閾値において幾つかのノードを選択した様子を示したものである。 また縦軸に表示されるノード数、横軸に各々のノードをとってプロットしたのが Figure 6.23から Figure 6.36である。多少のばらつきはあるが、数値的解析と同様に閾値を変化させることで、全体としての表示図形数をある範囲に制御できているのが見てとれる。

このように Fractal View により、図形数をある範囲に制御することが可能である。この結果、人間にとって図形の複雑さが減少し視認性が高まると同時に、表示図形数が削減されることで、システムの画面更新速度が速くなる。第 8 章の考察において、より性能の高いグラフィックス・ワークステーションによる VOGUE の実現について述べるが、いかに性能の高い計算機においても、表示図形数と画面更新速度の比例関係は変わらない。従って、ここで行なった図形数の制御は重要であると考えられる。

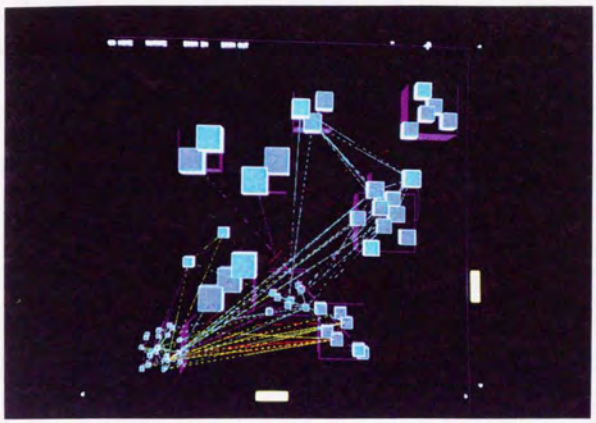


Figure 6.9: Fractal View による表示例

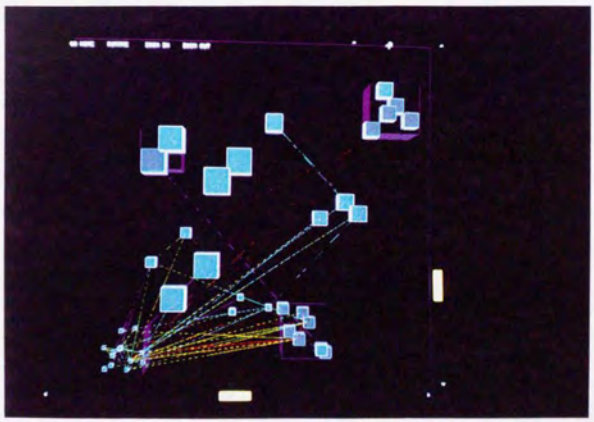


Figure 6.10: Fractal View による表示例

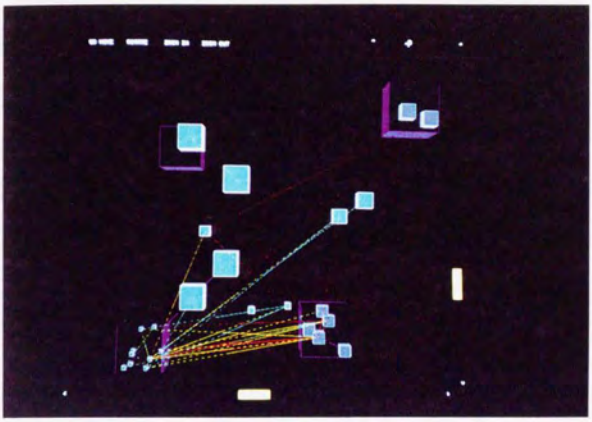


Figure 6.11: Fractal View による表示例

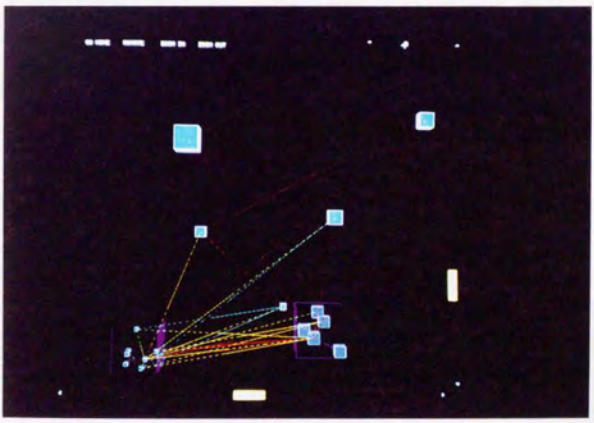


Figure 6.12: Fractal View による表示例

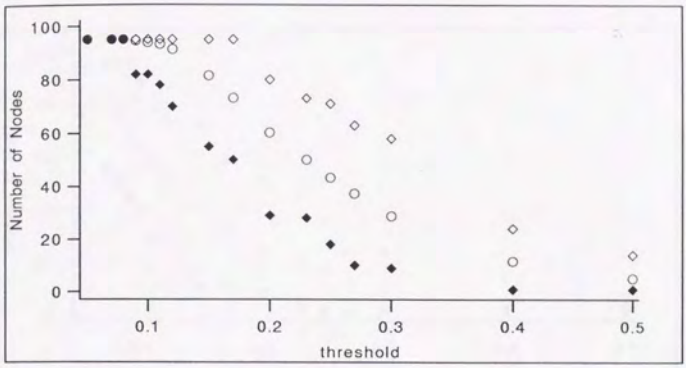


Figure 6.13: 閾値と表示されるノード数 (フラクタル次元 =3.5)

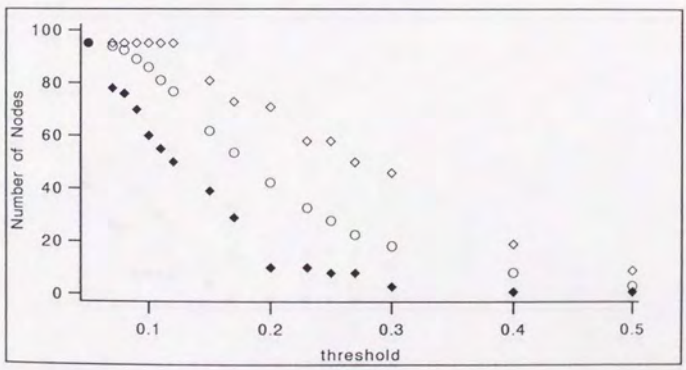


Figure 6.14: 閾値と表示されるノード数 (フラクタル次元 =3.0)

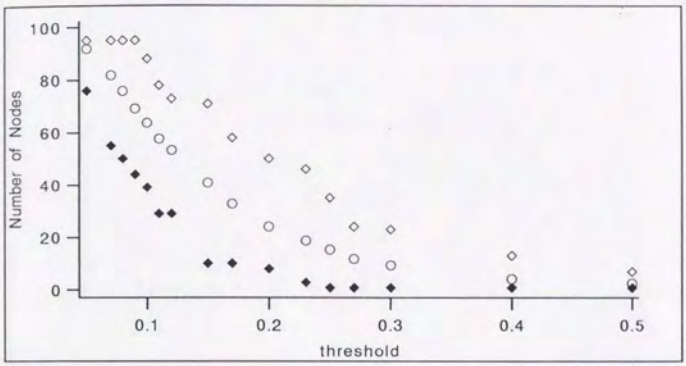


Figure 6.15: 閾値と表示されるノード数 (フラクタル次元 =2.5)

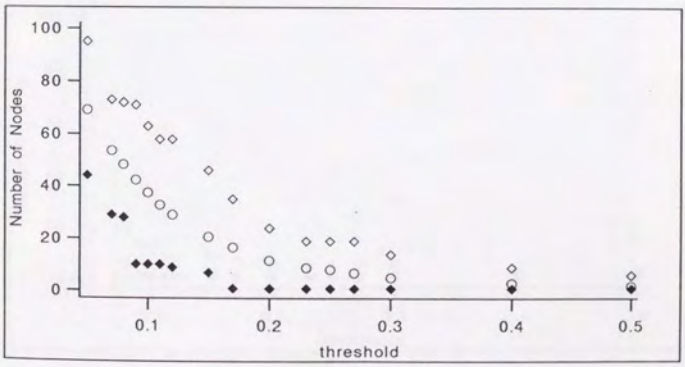


Figure 6.16: 閾値と表示されるノード数 (フラクタル次元 =2.0)

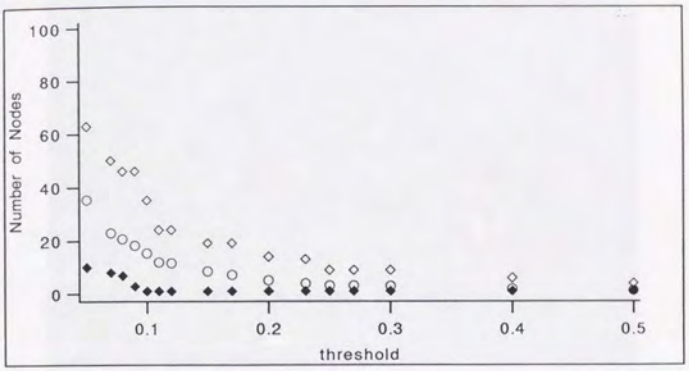


Figure 6.17: 閾値と表示されるノード数 (フラクタル次元 =1.5)

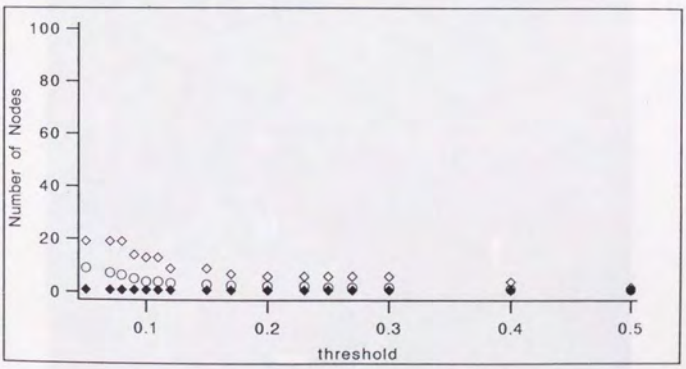


Figure 6.18: 閾値と表示されるノード数 (フラクタル次元 =1.0)

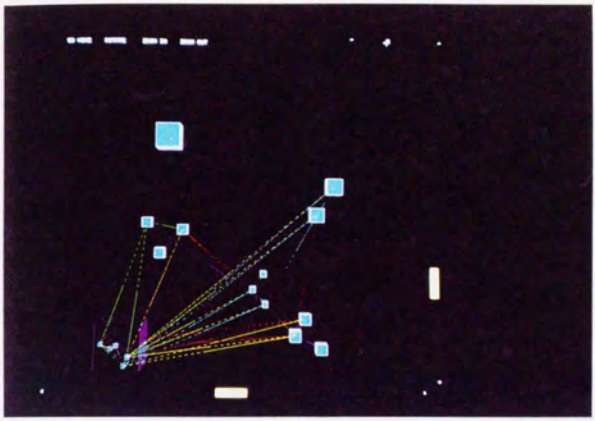


Figure 6.19: Fractal View の例

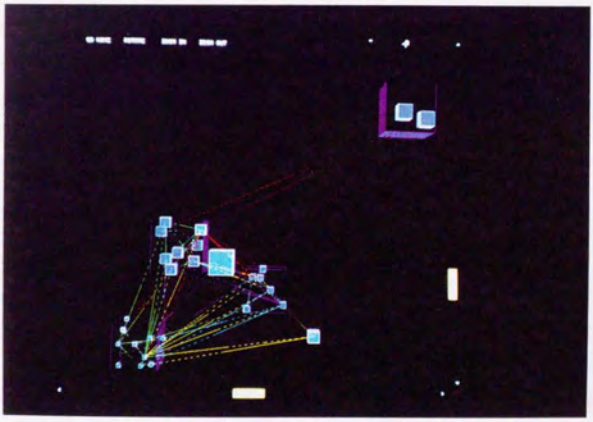


Figure 6.20: Fractal View の例

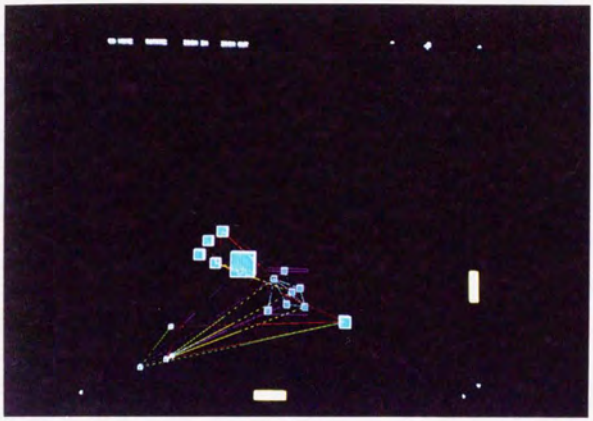


Figure 6.21: Fractal View の例

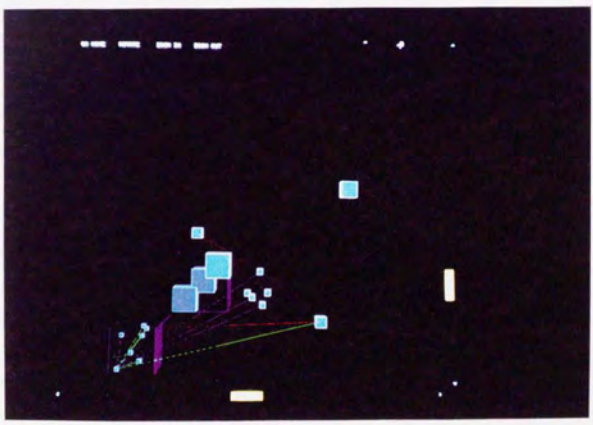


Figure 6.22: Fractal View の例

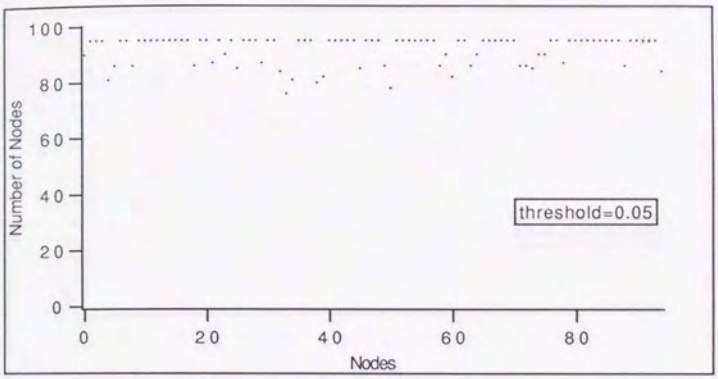


Figure 6.23: 閾値ごとのノード数の分布

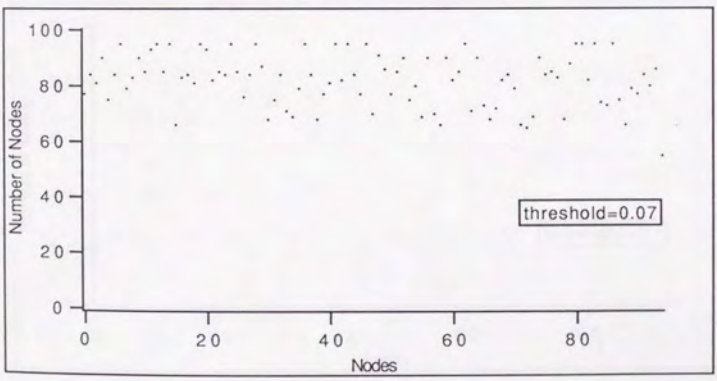


Figure 6.24: 閾値ごとのノード数の分布

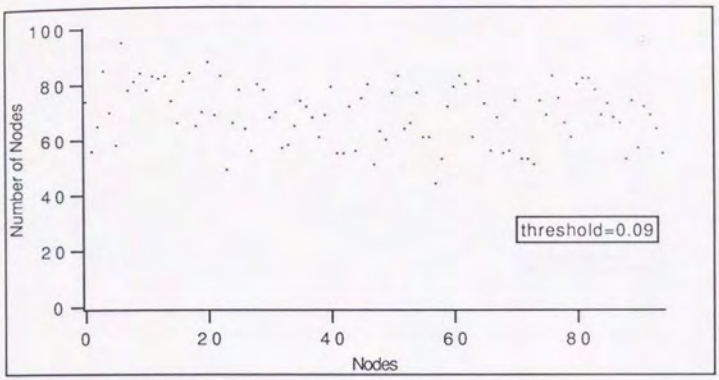


Figure 6.25: 閾値ごとのノード数の分布

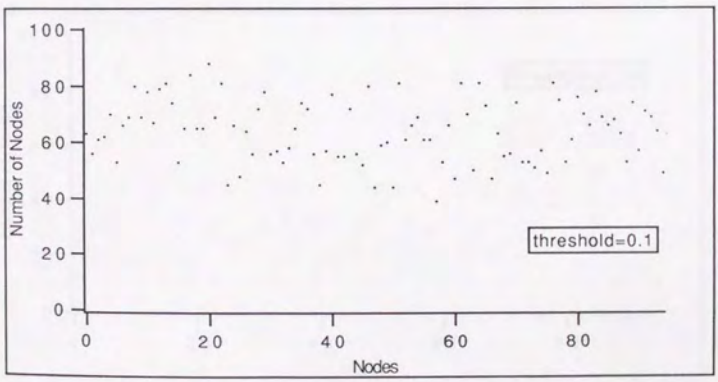


Figure 6.26: 閾値ごとのノード数の分布

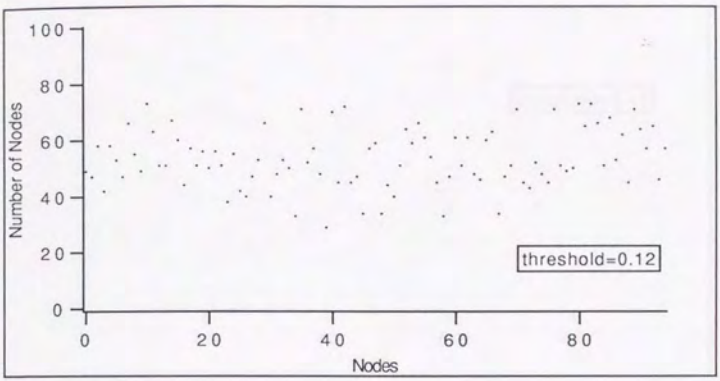


Figure 6.27: 閾値ごとのノード数の分布

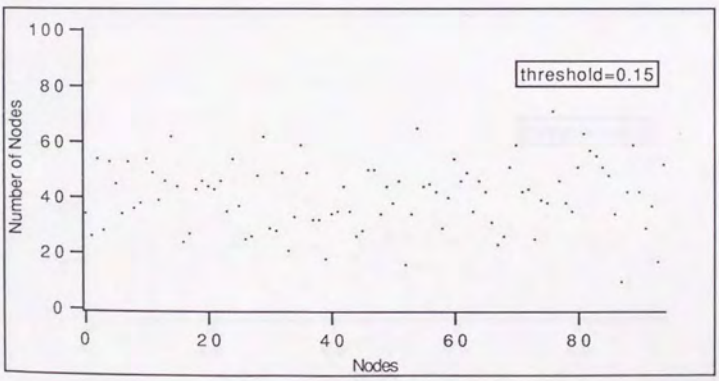


Figure 6.28: 閾値ごとのノード数の分布

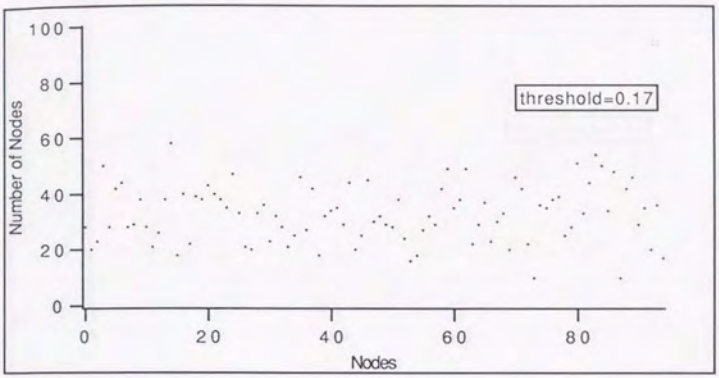


Figure 6.29: 閾値ごとのノード数の分布

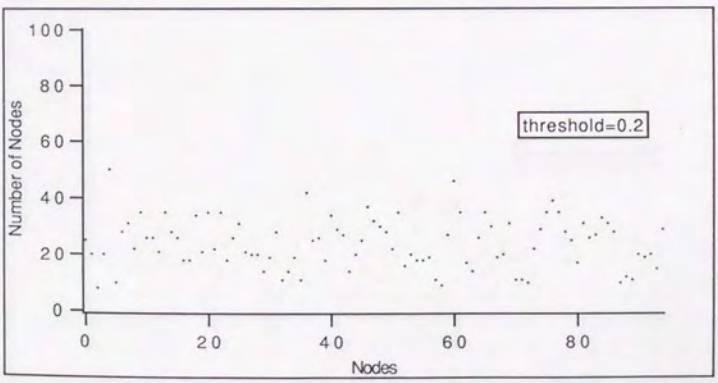


Figure 6.30: 閾値ごとのノード数の分布

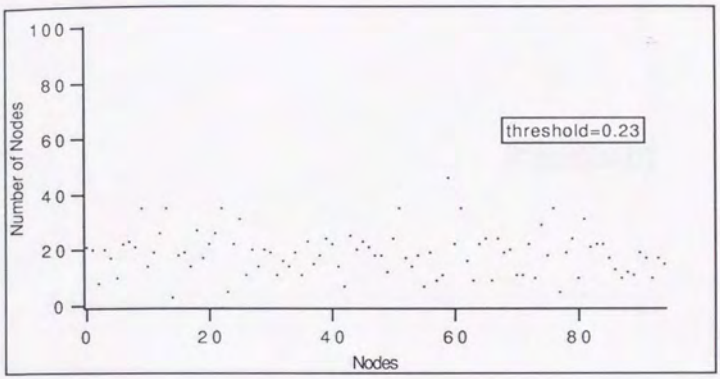


Figure 6.31: 閾値ごとのノード数の分布

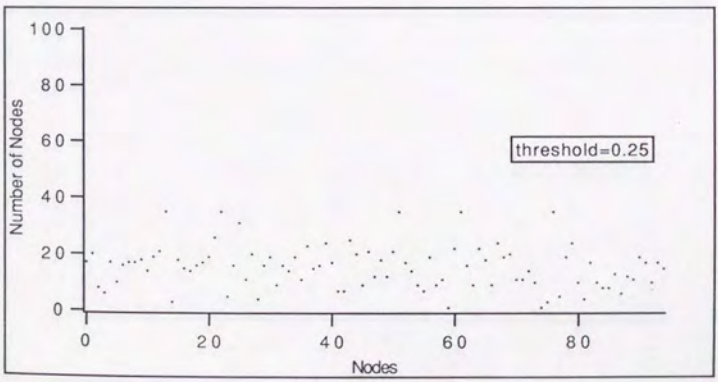


Figure 6.32: 閾値ごとのノード数の分布

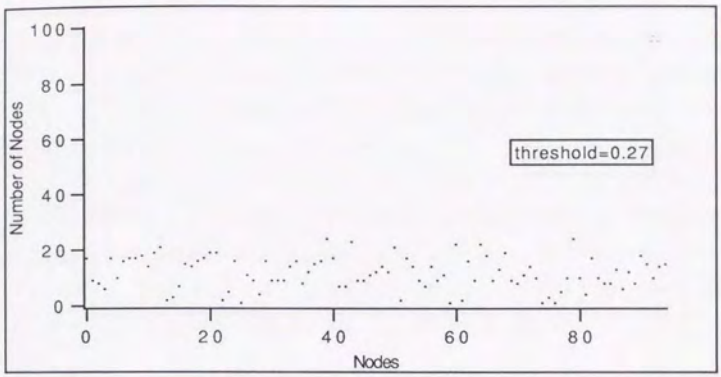


Figure 6.33: 閾値ごとのノード数の分布

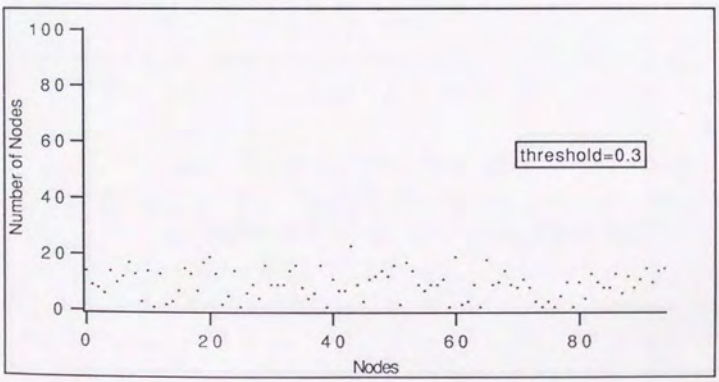


Figure 6.34: 閾値ごとのノード数の分布

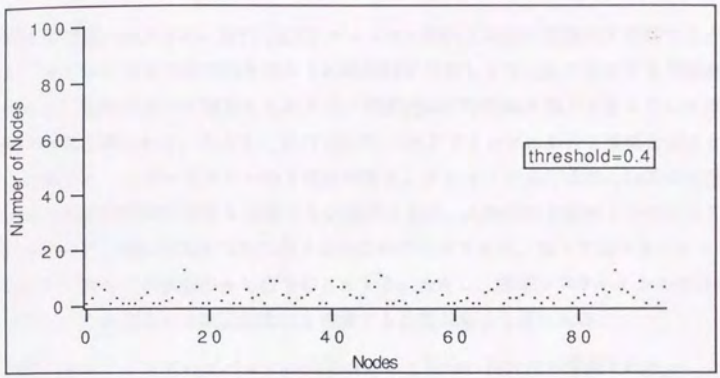


Figure 6.35: 閾値ごとのノード数の分布

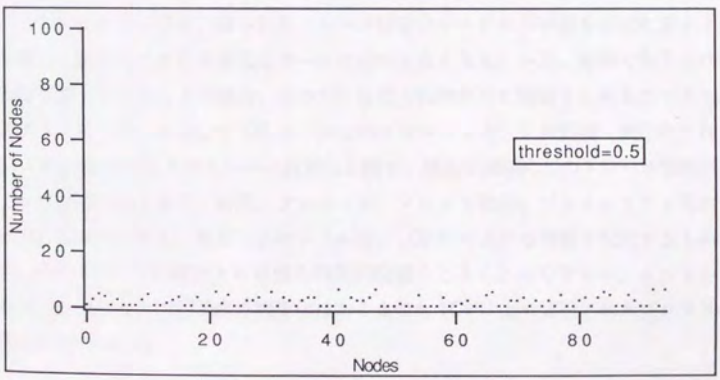


Figure 6.36: 閾値ごとのノード数の分布

6.5 時間軸の導入によるプロセス・モニタの実現

6.5.1 並行プロセスのデバッグと履歴の保存

第 3 章で述べたように、並行 (並列) プロセスの動的な挙動を視覚的に把握するためには、各プロセスの 2 次元的な広がりに時間軸を付加し 3 次元的に表現する方法が考えられる。前節において視覚化したタスク間関連図に時間軸を導入するとこの 3 次元表現が容易に得られる。ただし、並行 (並列) プログラミングにおける重要な視点として、制御フローとデータフローの 2 視点があり、プロセスの実行状態の時間的変化とともに資源の時間的変化をも考慮する必要があるが、本論文の主題の 1 つである 3 次元視覚化の有効性の実証を可能な限り単純な例で行なうため、以下で述べるプロセス・モニタはプロセスの状態のみを扱うこととする。ただし、実用システムにおいてはデータ・フローをも含めた 3 次元視覚化を考慮する必要があると思われる。

並列プログラミングのデバッグの問題点として動作の再現性が保証されないことがあげられる。本論文で実現したプロセス・モニタを他のシステムにおけるトレースの視覚化に応用することは容易であるが、一般的なオペレーティング・システム (OS) においてはプロセス状態の履歴保存機能は標準的に用意されていない。システム・コール等を利用して、これらを記録することは可能であるが、こうした履歴の検出は、第 3 章で述べたプローブ効果を増大させる危険性がある。OS レベルで支援された履歴保存が行なわれない限り、得られたトレース情報はシステムの挙動を正確に表すとは言い難く、従ってこうした視覚化ツールの意味もなくなる。一方、本章で取り上げた電力制御用ソフトウェアの場合、その OS は電力制御専用に開発されたものであり、専用デバッグ・ツールとして OS レベルでのプロセス・モニタが開発・実用化されている。Figure 6.37はそのトレース結果の 1 例で、現在は実際にこのトレース情報がデバッグに使用されており、時間、プロセス名、プロセス状態、プライオリティ等の情報を得ることができる。ただしこのツールは、1CPU における履歴を記録するものであり、他の CPU での履歴との正確な時間的関連をとることはできない。よってここで実現したプロセス実行状況の視覚化はあくまでも 1CPU 上で実行されたプロセスの状態 (state) である。

```
VAX/VMS  PERFORMANCE MONITOR V00.01   1-NOV-1990 01:02:58.25          PAGE =   1
 CONTEXT SWITCHING LOG DATA       <DEADLOCK-OK    >     25-OCT-1990 22:18:37.51

                INF.                          PROCESS                          SYSTEM
 NO.            TYPE          TIME      PID    NAME    STATE   PRI  M  EVENT      PC       PFC
 ----           -----         ----------  --------  ------  -----  ---  -  -----  --------  -----
    1           WAIT        22:18:18.54   200A9   _TTA7:  HIB     25  U         7FFEDF8A    584
    2           CTXST       22:18:18.54   10080   MMSKLT  CUR     24  U           D28D
    3           EVENT       22:18:18.58   100AA   DBBOBA  LEF   1 30     AST     7FFEDE09
    4           CTXED       22:18:18.58   10080   MMSKLT  CUR     24  K           17F80      0
    5           CTXST       22:18:18.58   100AA   DBBOBA  CUR     30  E         7FFEDE09
    6           EVENT       22:18:18.58   100AA   DBBOBA  CUR     30     EVENT   7FFEDE09
    7           EVENT       22:18:18.58   200AB   ERRFMT  LEF  27 27     AST     7FFEE128
    8           WAIT        22:18:18.58   100AA   DBBOBA  HIB     30  K         7FFEDF8A      0
    9           CTXST       22:18:18.58   200AB   ERRFMT  CUR     27  U         7FFEE128
   10           EVENT       22:18:18.58   1006E   FILAIP  HIB     24     EVENT   7FFEDF8A
   11           EVENT       22:18:18.58   1006E   FILAIP  HIB     24     AST     7FFEDF8A
   12           EVENT       22:18:18.58   200AB   ERRFMT  CUR     27     EVENT   7FFEE128
   13           EVENT       22:18:18.59   200AB   ERRFMT  CUR     27     EVENT   7FFEE128
   14           WAIT        22:18:18.59   200AB   ERRFMT  HIB     27  U         7FFEDF8A      0
   15           CTXST       22:18:18.59   10080   MMSKLT  CUR     24  K           17F80
   16           EVENT       22:18:18.59   1006C   RMSAIP  HIB     24     EVENT   7FFEDF8A
   17           EVENT       22:18:18.59   1006C   RMSAIP  HIB     24     AST     7FFEDF8A
   18           CTXED       22:18:18.59   10080   MMSKLT  CUR     24  U           11EF9A     0
   19           CTXST       22:18:18.59   1006E   FILAIP  CUR     24  U         7FFEDF8A
   20           EVENT       22:18:18.59   1006E   FILAIP  CUR     24     EVENT   7FFEDF8A
   21           EVENT       22:18:18.59   1006E   FILAIP  CUR     24     WAKE    7FFEDF8A
   22           WAIT        22:18:18.60   1006E   FILAIP  HIB     24  U         7FFEDF8A      0
   23           CTXST       22:18:18.60   1006C   RMSAIP  CUR     24  U         7FFEDF8A
   24           EVENT       22:18:18.60   1006C   RMSAIP  CUR     24     WAKE    7FFEDF8A
   25           EVENT       22:18:18.60   1006C   RMSAIP  CUR     24     EVENT   7FFEDF8A
   26           WAIT        22:18:18.60   1006C   RMSAIP  HIB     24  U         7FFEDF8A      0
   27           CTXST       22:18:18.60   10080   MMSKLT  CUR     24  U           11EF9A
   28           EVENT       22:18:18.66   1009E   TEST    LEF   1 29     EVENT   7FFEE07A
   29           EVENT       22:18:18.66   1006F   QPSAIP  LEF  16 27     AST     7FFEDE09
   30           CTXED       22:18:18.66   10080   MMSKLT  CUR     24  U           E0B2       0
   31           CTXST       22:18:18.66   1009E   TEST    CUR     29  U         7FFEE07E
   32           WAIT        22:18:18.68   1009E   TEST    LEF  31 29  U         7FFEDE09      0
   33           CTXST       22:18:18.68   1006F   QPSAIP  CUR     27  U         7FFEDE09
   34           WAIT        22:18:18.68   1006F   QPSAIP  LEF  16 27  U         7FFEDE09      0
   35           CTXST       22:18:18.68   10080   MMSKLT  CUR     24  U           E0B2
   36           EVENT       22:18:18.70   10025   JIKANN  HIB     22     WAKE    7FFEDF8A
   37           WAIT        22:18:18.72   10080   MMSKLT  PFW     24  K           17F80      1
   38           CTXST       22:18:18.72   10048   JYOKEN  CUR     23  U         7FFEDF8A
   39           WAIT        22:18:18.72   10048   JYOKEN  HIB     23  U         7FFEDF8A      0
   40           CTXST       22:18:18.72   10025   JIKANN  CUR     22  U         7FFEDF8A
   41           EVENT       22:18:18.73   10080   MMSKLT  PFW     24     PFCOM     17F80
   42           CTXED       22:18:18.73   10025   JIKANN  CUR     22  U         7FFEDF8A      0
   43           CTXST       22:18:18.73   10080   MMSKLT  CUR     24  K           17F80
   44           EVENT       22:18:18.73   100A6   NTAACP  HIB     30     AST     7FFEDF8A
   45           CTXED       22:18:18.73   10080   MMSKLT  CUR     24  K           17D50      0
   46           CTXST       22:18:18.73   100A6   NTAACP  CUR     30  K         7FFEDF8A
   47           EVENT       22:18:18.74   1006F   QPSAIP  LEF  16 27     AST     7FFEDE09
   48           WAIT        22:18:18.74   100A6   NTAACP  HIB     30  K         7FFEDF8A      0
   49           CTXST       22:18:18.75   1006F   QPSAIP  CUR     27  U         7FFEDE09
   50           EVENT       22:18:18.75   1006F   QPSAIP  CUR     27     EVENT   7FFEDE09
```

Figure 6.37: プロセス実行状態のトレース結果 (1 部抜粋)

1	COLP	*
2	MWAIT	*
3	CEF	*
4	PFW	マゼンタ
5	LEF	緑
6	LEFO	*
7	HIB	黄色
8	HIBO	*
9	SUP	*
10	SUPO	*
11	FPG	*
12	COM	シアン
13	COMO	*
14	CUR	赤

Table 6.2: プロセスの種類と VOGUE における色 (* は青)

6.5.2 VOGUE による実現方式

前節のトレースデータを用いて、本節ではプロセス実行状態の視覚化を行なう。トレースファイルは、時間、コンテキスト・スイッチング、プロセス ID 等の各情報を持っているが、ここでは簡単化のため行番号とプロセスの状態だけに注目する。従って、実現されるプロセス・モニタの奥行きは必ずしも正確な時間関係を表さないが、ここでの目的が正確な時間関係の把握ではないので許容されうる。ただし、実用化システムを構築する場合には当然考慮されなければならない。また、プロセスの状態には Table 6.2に示す 14 の状態があるが、ここではトレースファイルに頻繁に出現する 5 つにだけ注目することにし、これらを異なる色のノードとして表示する。

実現にあたってはまず、7.2 節で実現したタスク間通信図を xy 平面上に配置し、各タスクに対応するノードの形状を z 軸方向に伸びた細い柱状に変更する。これは、プロセス状態を表すノードがどのタスクの状態を表すか認識しやすくするためである。次に、Figure 6.37に示したトレース・ファイルを一行ずつ読んで、対応するタスクの x、y 座標、および行番号に対応する z 座標にプロセスに対応するノードを配置す

る。このノードはデータベースにおいて add-object で生成されるのではなく、直接 graph-node のインスタンスとして生成され、グラフ g に登録される。トレースデータの各行に相当するインスタンスをデータベースに作成することは可能であるが、インスタンス間の順序関係等の管理が繁雑になるため、このような手法をとった。またプロセス・ノードだけを表示するとプロセス間の順序関連が把握し難くなるため、直前まで CUR 状態だったノードから現在 CUR 状態のノードに対してリンクを張ることにする。実際にトレースファイルから視覚化表現を得る手続きの概要を Figure 6.38に掲載する。

Figure 6.39および Figure 6.40がこうして VOGUE で実現されたプロセス・モニタである。真横から見ると従来の 2 次元上でのプロセス・モニタになり、xy 平面に垂直な方向から見るとプロセスの 2 次元的広がりを把握することができる。以降の節では、このプロセス・モニタを利用して実際の電力制御システムの障害事例を視覚化し、視覚表現の有効性と 3 次元化の意義について述べる。

```
;; 指定されたトレースファイルを開く
(with-open-file (st    トレースファイル名)
 ;; ファイルから 1 行読み込む
 (do ((line (read-line st nil nil) (read-line st nil nil))
          :  )
     ((null line))
          :
     行番号、プロセス名、ステートを得る処理
          :
     ;; グラフから label がプロセス名のノード、つまりタスク
     ;; に対応するノードを検索する
     (let ((node (assoc-node グラフ 'label プロセス名)))
        (when node
              ;; graph-node のインスタンスを 1 個生成する。ただし上で
              ;; x,y 座標は検索したタスクノードに等しく、z 座標は
              ;; 適当な位置、色はステートから決定する
              (let ((gnode (make-instance 'graph-node
                                :xpos (xpos node)
                                :ypos (ypos node)
                                :zpos counter
                                :length 0.1
                                :color ステートに対応する色)))
                ;; z 座標のカウンタをインクリメント
                (incf counter)
                     :
                カレントノードにリンクを張る処理
                     :
                ;; グラフに作成したノードを登録して表示
                (add-node グラフ gnode)
                (display gnode)
                ;; 画面更新
                (refresh *hp*))))))
```

Figure 6.38: プロセス・モニタの処理の概要

Figure 6.39: VOGUE によるプロセス・モニタの実現 (正面図)

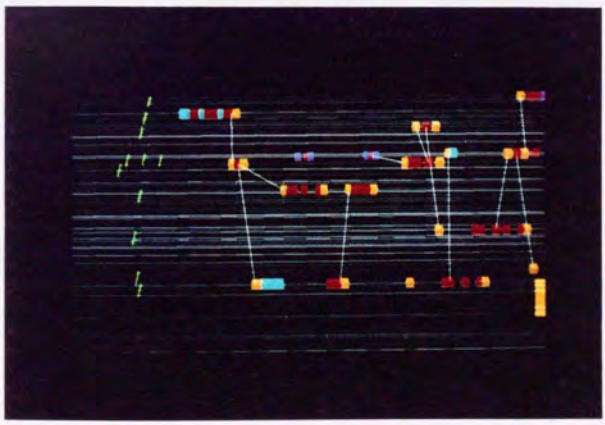


Figure 6.40: VOGUE によるプロセス・モニタの実現 (側面図)

6.6 電力制御用システム障害時の実例の視覚化

本節では、電力システムにおける障害事例を対象として、作成したプロセス・モニタによる視覚化を行ない視覚化ツールがいかにデバッグに利用できるかについて述べる。ただし、実際の障害が起こった時のトレースデータを得るのは現実には困難であり、ここでのトレースデータは何らかのパッチをあてることで、実際に起こった障害事例を疑似的に再現したものである。しかし、トレースデータを取得した計算機は現実に稼働しているものと全く同じ条件のシステムであり、実機において同様な障害が発生した場合には同じような視覚化表現が得られると考えられる。

6.6.1 デッドロック

電力制御システムの制御所のマン・マシン・インタフェースとして、各情報を表示する幾つかの画面があるが、そのうちの一つに各発電所、変電所等をグラフとして表示し現在の運転状況を監視するスケルトンと呼ばれる画面がある。これに関連するプロセスとしてはスケルトン表示プロセス (MMSKLT) と OP-DISP 表示プロセス (MMOPEC) がある。そしてこれらに関係する資源として表示ステータス (DPS) と CRT 画面 (CRT) がある。いま、スケルトン画面の 5 秒周期の計測値再表示処理と、画面消去のタイミングが重なると、スケルトン表示プロセスと OP-DISP 解析プロセスの間で、資源の取り合いが発生する (Figure 6.41)。

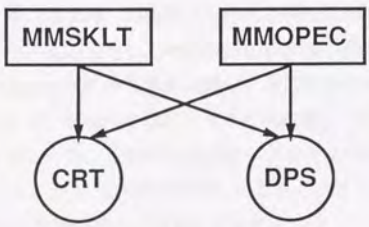


Figure 6.41: 資源の取り合い

通常は、資源確保の順番を DPS、CRT と決めているのでデッドロックは起こらないが、MMOPEC の資源確保の順番を通常とは逆の CRT、DPS とした場合のデッ

ドロックを想定する。Figure 6.42は正常時、Figure 6.43はデッドロック時のタイミングチャートである。

Figure 6.44がトレース・データの 1 部を抜粋したものであるが、先にも述べたようにここでは資源の状態に対するトレースがないので、プロセスの状態だけを表示する。このトレース・データの実際の大きさは以下に示すとおりで、開発者はこの膨大なトレース・データを基に、個人的ノウハウを利用してシステムのバグを同定していく。

状態	ページ数	行数	バイト数
正常時	89	4444	578032
異常時	89	4444	578032

一方、Figure 6.45、Figure 6.46が正常時の VOGUE による視覚化の正面図および側面図、Figure 6.47、Figure 6.48がデッドロック時の正面図および側面図である。視認性を高めるため MMSKLT 以外のタスクに対応する柱は消してある。

また Figure 6.49はこれら 2 つを各々別なグラフとして生成し、同時に 1 画面で比較した例である。VOGUE ではモデルの実体、つまりインスタンス同士の関係とグラフとを別なものとして扱っているため、このような機能が実現できる。また、各タスクの位置は save-graph で一度保存したものを各々別なグラフに再ロードしているため正確に一致している (ただし、この場合画面更新速度は極端に遅くなっている)。

全体的な印象としては、正常時、障害時ともそれ程差は認められない。特に前半部分はほとんど同じ形状である。しかし、中間部分から両者の形に差がでているのがわかる。Figure 6.45と Figure 6.47の写真を比較し、特に柱状のタスクが表示されている MMSKLT に注目すると、障害時にはプロセスの色が緑、つまり LEF 状態で止まったまま沈黙に入っている。一方、正常時には何度がプロセスが走り最終的に黄色、つまり HIB で終了している。プロセスが資源待ちになると、ステートは LEF になって止まるので[2]、MMSKLT は資源待ちの状態のままデッドロックに入ったことがわかる。

[2]ただし、デバイスの I/O 待ちの場合なども LEF となるため、LEF＝資源待ちとは限らない

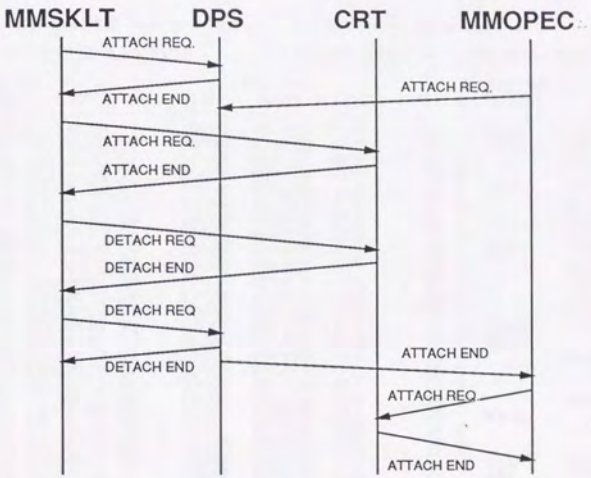


Figure 6.42: 平常時のタイミングチャート

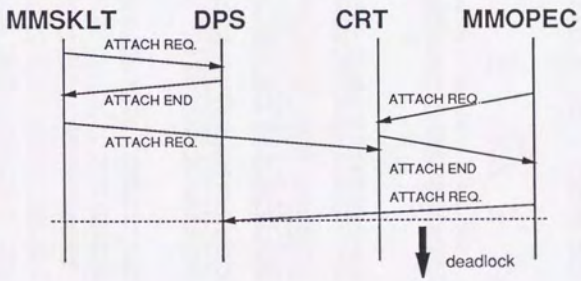


Figure 6.43: デッドロック時のタイミングチャート

```
VAX/VMS  PERFORMANCE MONITOR V00.01   1-NOV-1990 16:58:36.10        PAGE =  68
 CONTEXT SWITCHING LOG DATA        <DEADLOCK-BUG   >    25-OCT-1990 22:18:37.51

 NO.  INF. TYPE    TIME          PID      PROCESS NAME  STATE   PRI  M  SYSTEM EVENT    PC        PFC
----  ---------- ------------ --------  ------------  -----  ---  -  ------  --------  -----
3351  EVENT  22:18:32.27     10081   MMOPEC  COM      24     WAKE    7FFEDF8A
3352  EVENT  22:18:32.27     100A6   NTAACP  HIB      30     WAKE    7FFEDF8A
3353  CTXED  22:18:32.27     10086   MMPBIN  CUR      25  K          80008956      0
3354  CTXST  22:18:32.27     100A6   NTAACP  CUR      30  K          7FFEDF8A
3355  WAIT   22:18:32.27     100A6   NTAACP  HIB      30  K          7FFEDF8A      0
3356  CTXST  22:18:32.27     10086   MMPBIN  CUR      25  K          80008956
3357  WAIT   22:18:32.28     10086   MMPBIN  HIB      25  U          7FFEDF8A      0
3358  CTXST  22:18:32.28     10081   MMOPEC  CUR      24  U          7FFEDF8A
3359  EVENT  22:18:32.28     100A6   NTAACP  HIB      30     AST     7FFEDF8A
3360  CTXED  22:18:32.28     10081   MMOPEC  CUR      24  U          7FFEDF8A      0
3361  CTXST  22:18:32.28     100A6   NTAACP  CUR      30  K          7FFEDF8A
3362  WAIT   22:18:32.28     100A6   NTAACP  HIB      30  K          7FFEDF8A      0
3363  CTXST  22:18:32.28     10081   MMOPEC  CUR      24  U          7FFEDF8A
3364  EVENT  22:18:32.29     10081   MMOPEC  CUR      24     EVENT   7FFEDF8A
3365  EVENT  22:18:32.29     10085   MMLPIN  HIB      25     WAKE      9C2A
3366  CTXED  22:18:32.29     10081   MMOPEC  CUR      24  K          8000A07F      0
3367  CTXST  22:18:32.29     10085   MMLPIN  CUR      25  U            9C2A
3368  WAIT   22:18:32.30     10085   MMLPIN  HIB      25  U            9C2A        0
3369  CTXST  22:18:32.30     10081   MMOPEC  CUR      24  K          8000A07F
3370  EVENT  22:18:32.30     10084   MMPBOU  HIB      26     WAKE      36A0A
3371  CTXED  22:18:32.30     10081   MMOPEC  CUR      24  K           4A13F      0
3372  CTXST  22:18:32.30     10084   MMPBOU  CUR      26  U            36A0A
3373  EVENT  22:18:32.31     1006F   QPSAIP  HIB      27     EVENT   7FFEDF8A
3374  EVENT  22:18:32.31     1006F   QPSAIP  HIB      27     AST     7FFEDF8A
3375  CTXED  22:18:32.31     10084   MMPBOU  CUR      26  U             D17       18
3376  CTXST  22:18:32.31     1006F   QPSAIP  CUR      27  U          7FFEDF8A
3377  WAIT   22:18:32.31     1006F   QPSAIP  HIB      27  U          7FFEDF8A      0
3378  CTXST  22:18:32.32     10084   MMPBOU  CUR      26  U             D17
3379  EVENT  22:18:32.32     100A6   NTAACP  HIB      30     WAKE    7FFEDF8A
3380  CTXED  22:18:32.32     10084   MMPBOU  CUR      26  K          80008956     11
3381  CTXST  22:18:32.32     100A6   NTAACP  CUR      30  K          7FFEDF8A
3382  WAIT   22:18:32.33     100A6   NTAACP  HIB      30  K          7FFEDF8A      0
3383  CTXST  22:18:32.33     10084   MMPBOU  CUR      26  K          80008956
3384  WAIT   22:18:32.33     10084   MMPBOU  LEF   0  26  U          7FFEDE09      0
3385  CTXST  22:18:32.33     10081   MMOPEC  CUR      24  K            4A13F
3386  EVENT  22:18:32.33     100A6   NTAACP  HIB      30     AST     7FFEDF8A
3387  CTXED  22:18:32.33     10081   MMOPEC  CUR      24  U            1380E       0
3388  CTXST  22:18:32.33     100A6   NTAACP  CUR      30  K          7FFEDF8A
3389  WAIT   22:18:32.33     100A6   NTAACP  HIB      30  K          7FFEDF8A      0
3390  CTXST  22:18:32.33     10081   MMOPEC  CUR      24  U            1380E
3391  EVENT  22:18:32.34     10082   MMANNA  HIB      23     WAKE      3E02A
3392  EVENT  22:18:32.34     10081   MMOPEC  CUR      24     EVENT     1380E
3393  EVENT  22:18:32.34     100A6   NTAACP  HIB      30     AST     7FFEDF8A
3394  CTXED  22:18:32.34     10081   MMOPEC  CUR      24  K          8012A030      0
3395  CTXST  22:18:32.34     100A6   NTAACP  CUR      30  K          7FFEDF8A
3396  EVENT  22:18:32.34     10084   MMPBOU  LEF   0  26     AST     7FFEDE09
3397  WAIT   22:18:32.34     100A6   NTAACP  HIB      30  K          7FFEDF8A      0
3398  CTXST  22:18:32.34     10084   MMPBOU  CUR      26  U          7FFEDE09
3399  EVENT  22:18:32.35     10084   MMPBOU  CUR      26     EVENT   7FFEDE09
3400  EVENT  22:18:32.37     100A6   NTAACP  HIB      30     AST     7FFEDF8A
```

Figure 6.44: デッドロック時のトレースデータ (一部抜粋)

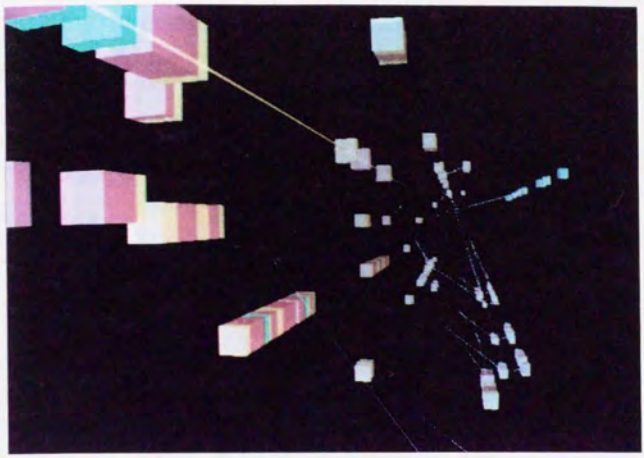


Figure 6.45: 正常時 (正面図)

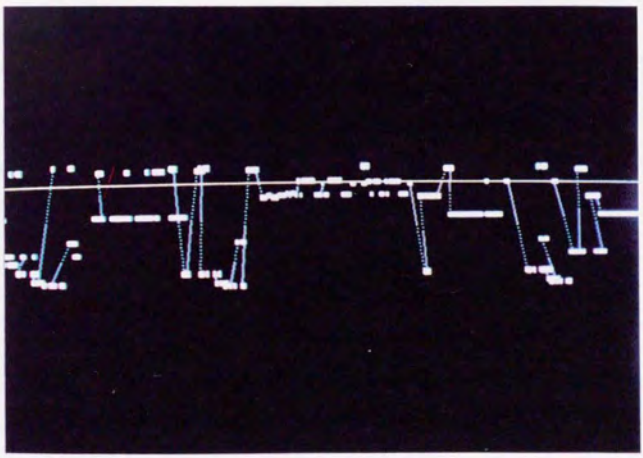


Figure 6.46: 正常時 (側面図)

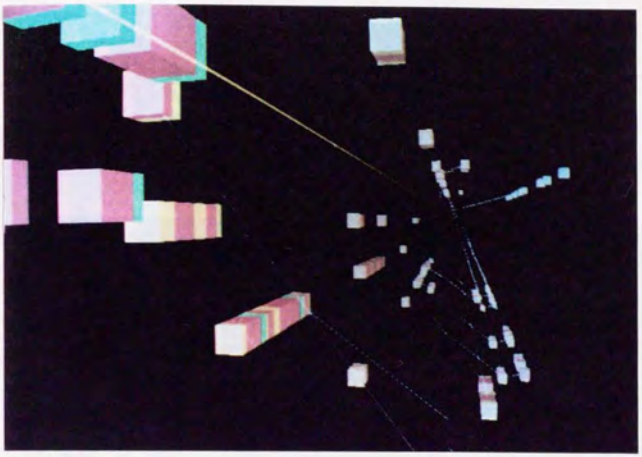


Figure 6.47: デッドロック時 (正面図)

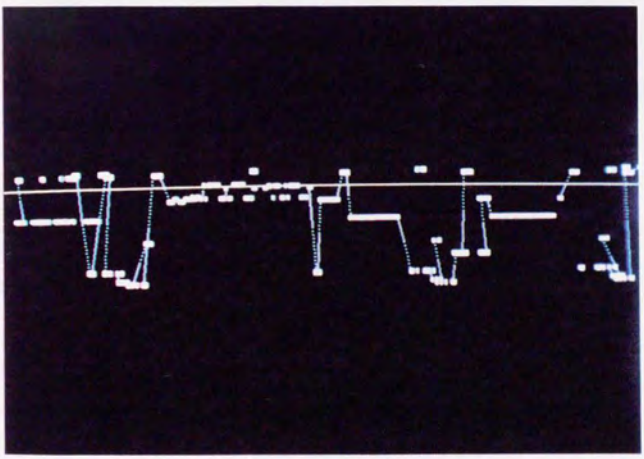


Figure 6.48: デッドロック時 (側面図)

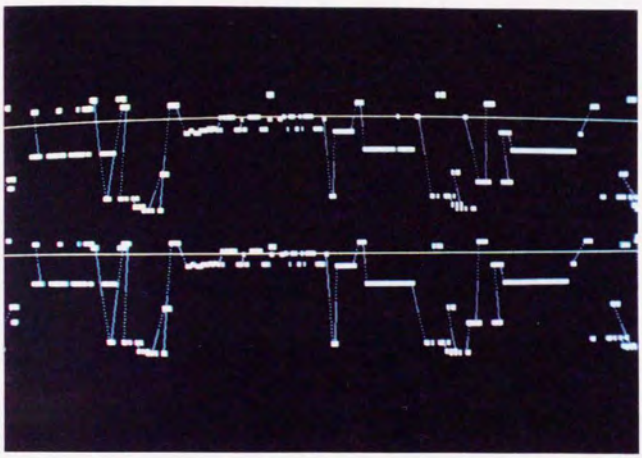


Figure 6.49: 正常時とデッドロック時の比較

6.6.2 無限ループ

次に、同じ制御用計算機の実機を用いて無限ループ状態を仮作成し、その視覚化を行なった。電流の監視に使用するデータは普段は Figure 6.50左のような、線形リスト構造をなしているが、これを右のようにループになるようにすることで、疑似的に無限ループの状態を作成した。監視データがループになっていると、電流電圧監視プロセス (JYVICH) が無限ループを引き起こす。

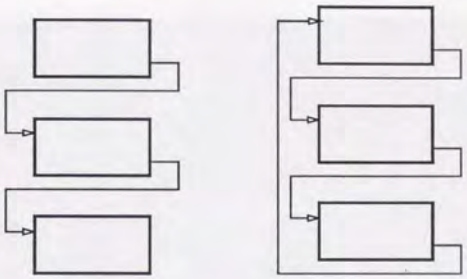


Figure 6.50: 監視データ

この状態で実機を走らせ取得したトレースファイルは以下の大きさで、現在はこのトレースファイルから関連すると思われるプロセスの抽出、正常時との比較等の作業を行なっている。

状態	ページ数	行数	バイト数
正常時	34	1692	220752
異常時	68	3383	441616

一方、このトレースファイルを基に VOGUE で視覚化を行なったものが Figure 6.51である。平常時には、制御が状変解析プロセスから電流電圧監視プロセスに移り、線形リストになったデータの監視が終了すると、再び制御が状変解析プロセスに移って終了する。一方、障害時の場合にはこの電流電圧監視プロセスがいつまでたっても終了しないのが、視覚的にも明かである。

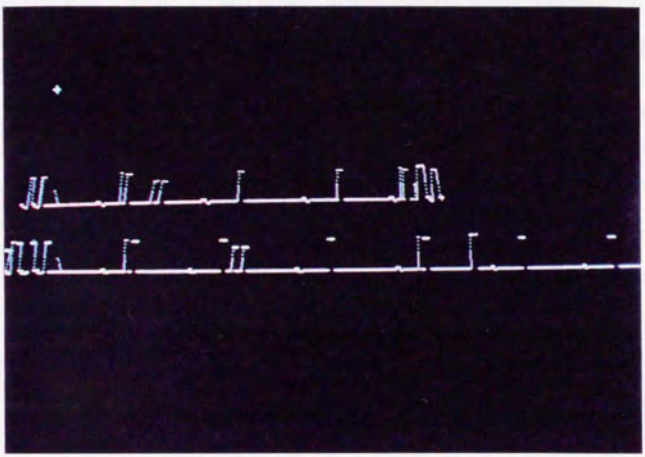


Figure 6.51: 無限ループの視覚化

6.6.3 プロセス・モニタに関する考察

さて、実際のデバッグ時の処理について考えてると、先にも述べたように現在デバッグの手法は、開発者のノウハウによるところが大きく、トレースデータから可能性の高いと思われる部分を抽出し、詳しく調べるという処理が行なわれている。その処理はボトムアップ的な作業ということができる。現実の障害発生時にトレースデータを取得した段階では、まだデッドロックあるいは無限ループといった障害の種別および障害発生プロセスは明らかではない。一方、視覚的に表現した場合には、まずマクロな見地からトレースを見て、障害が発生した可能性の高いプロセスの同定、あるいはその種別を正常時の視覚化と比較することによって行なうことができる。先の例で言えば無限ループの存在を認識した上で、その対象を絞りこんでゆく作業はトップダウン的な処理となる。従来のデバッグ手法にはこのようなトップダウン的な支援が不足していたと考えられる。特にシステムの挙動が把握しにくく、同定すべき障害の要素の多い大規模なソフトウェアのデバッグにおいては、このようなトップダウン的支援が重要となると考えられる。これは図の把握が全体から部分に進む特徴を生かした例といえる。

次に、図形削減について考える。実現したプロセス・モニタは、基本的にタスクを表現する柱に対してプロセス状態を表現するノードを並べているだけなので、本例では Fractal View や木の深さに着目した削減手法は使えない。本例では型に着目した次のような削減手法が適用できる。各プロセスの状態に対応するノードの作成時に figure スロットに対応するタスクのインスタンスを入れる。focus-node はノードの figure スロットに入っているインスタンスの型を判断するので、これにより注目するタスクだけを表示することが可能となる。

試作したプロセス・モニタは 2 次元でも十分実現可能であると思われる。特に、現在のようにプロセス間をつなぐリンクがあまり重要な意味を持たず、プロセスの状態だけに着目している場合には 3 次元化の意義は小さい。3 次元化が重要性を増してくるのは、次節で述べるようにプロセス間のメッセージ送信が視覚化された場合であると考えられるが、本節の単純なプロセス・モニタにおいても 3 次元視覚化の有効性を述べることはできる。

まず第 1 に、Figure 6.46と Figure 6.48のような側面図は、現在の 2 次元平面上でのプロセス・モニタに対応しているが、グラフの全体像を把握するためには、視点の位置を十分後方にとらなくてはならない。この時、ノードの色を認識することはもは

や不可能である。また、プロセスの詳細を調べるために接近すると、今度はグラフ全体としての形が把握できなくなってしまう。これに対して、Figure 6.45とFigure 6.47のような角度からグラフを見ると、視点位置の近傍のノードは色まではっきりと認識でき、かつグラフの後方の状態をも同時に把握することが可能である。もちろん前者の場合、横方向の縮尺を変更して、より短い範囲内に全グラフを表示することで対応できそうであるが、この場合には1ノードの奥行きが極端に短くなり、微妙な色の変化を認識するのに障害がおこると考えられる。また、次節で述べるメッセージ送信の様子を視覚化する場合には、リンク同士の間隔が狭くなり、見づらいグラフとなる。

第2に、2次元的表示の場合にはサブシステム間の関連が把握しにくい点が挙げられる。Figure 6.52とFigure 6.53は、別なトレースデータの視覚化であるが、後者の図からはサブシステム間の関係はわからない。一方、前者を見ると、これは明らかに3つのサブシステムの相互動作であることが理解できる。本来、空間的広がりのあるモジュール間の関係を、2次元プロセス・モニタでは1軸上で表現しなければならない。しかし3次元視覚化を行なうことで、これらを空間的パターンとして自然に表現することが可能になるのである。この柱状に表現されたプロセス間を策走するリンクの視認性に関して、次のような実験がなされている [47]。

今、例えばプロセスが2つの場合には、2次元表示であろうと3次元表示であろうと差はでないと思われる。一方、最低3つのプロセスがある場合2次元と3次元では把握のしやすさに差がでる。Figure 6.54のようなパターンの場合CからAへのリンクがBを横切らざるを得ないためである。よって実験では、あるプロセスから2つのプロセスを通って、また元のプロセスに戻ってくるようなループを題材とした。Figure 6.55のように3つのプロセスの柱に対し、その柱と柱との間にメッセージを示す矢印をランダムに表示する。矢印には赤と青があり、色が異なれば別なパターンを示すものとする。

被験者に対してこの画面を提示し、被験者はループの有無を判断する。そして画面の提示から判断までの時間を記録する。時間が早いほど認識しやすいといえる。それぞれ結果はTable 6.3のようになる。明らかに3次元の方が見やすいことが判る。特に2次元表示の場合、ループが存在しないとき、それを認識するのが困難である。

この実験結果から言えることは、プロセスモニタを2次元で実現することは、可能であるが、もともとプロセス間通信は2次元的な広がりを持っているため、時間軸を加えて3次元に表示することで、よりユーザの認知が早く正確になるということであ

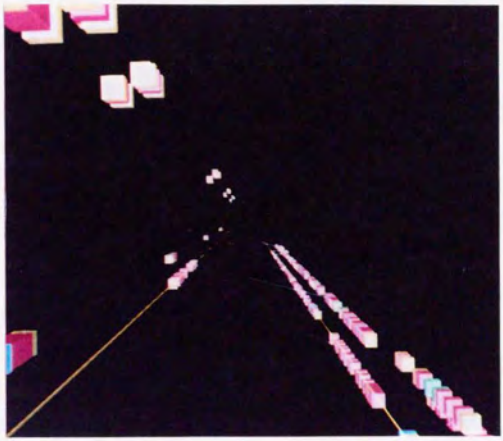


Figure 6.52: 他の障害事例の視覚化 (正面図)

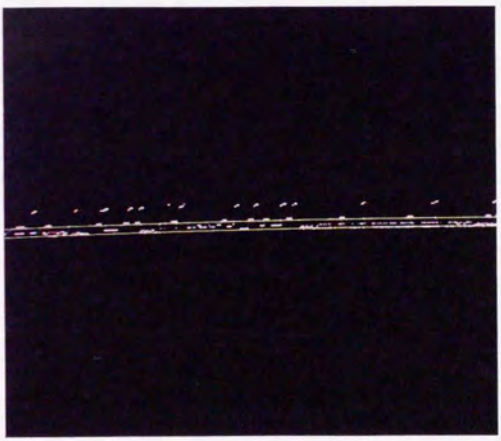


Figure 6.53: 他の障害事例の視覚化 (側面図)

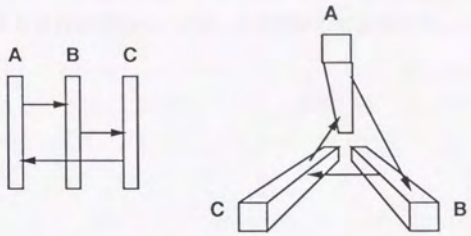


Figure 6.54: 3 プロセス間のメッセージ送信

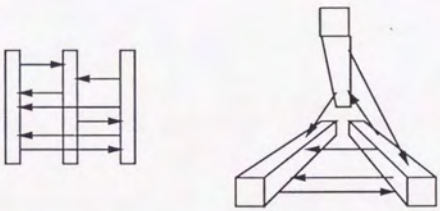


Figure 6.55: 実験で用いた画面例

	ループのある場合	ループのない場合	全体
2D	7	15	12
3D	7	7	7

Table 6.3: 実験結果

る。

本節におけるリンクには意味がないことは先にも述べたが、次節ではこのリンクがプロセス間、あるいは計算機間のメッセージ送信として意味を持ってくる場合について述べる。

6.7 分散マニピュレータにおけるメッセージ送信の視覚化

6.7.1 分散マニピュレータ

電力制御システムにおける各計算機間のメッセージ送信の記録が現段階では不可能であることは前にも述べたが、将来的にこうした複数台計算機間のメッセージ送信が記録できるようになった時、3 次元視覚化がいかに利用されるかを示すために、本節では自律分散型マニピュレータにおけるメッセージ送信のトレースに基づき考察する。

自律分散型マニピュレータとは、複数の回転型関節を直列に接続した多関節型ロボットアームであり、各関節ごとに各々計算機を組み込んで搭載している点が特徴である。この自律分散型マニピュレータの特徴としては、

- マニピュレータ内部に引き回されるワイヤハーネスの合理化が可能となる
- 分散された各要素の冗長性をうまく利用することでシステムの高信頼化が図れる
- ソフトウェア開発の効率化

があげられる。Figure 6.56が対象としたマニピュレータの写真で、Figure 6.57がその概念図である。

各計算機は基本的に並列に動作し、メッセージ送信という形態で他の計算機と通信を行なうという協調動作のもとで与えられたタスクを実行する。よって、分散並列処理システムの典型的な例であり、ここで得られる知見はそのまま他の分散並列システムにも適用しうるものであると考えられる。

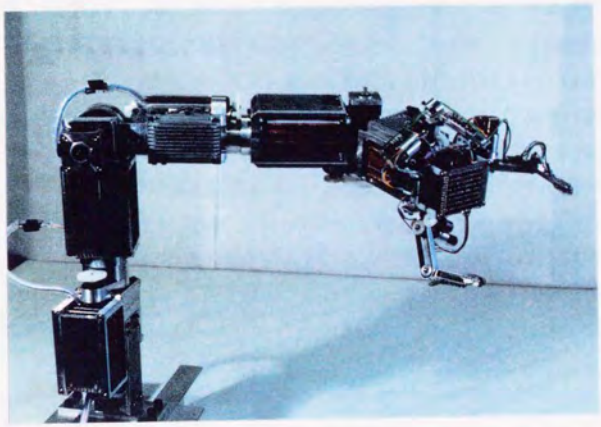


Figure 6.56: 分散マニピュレータ

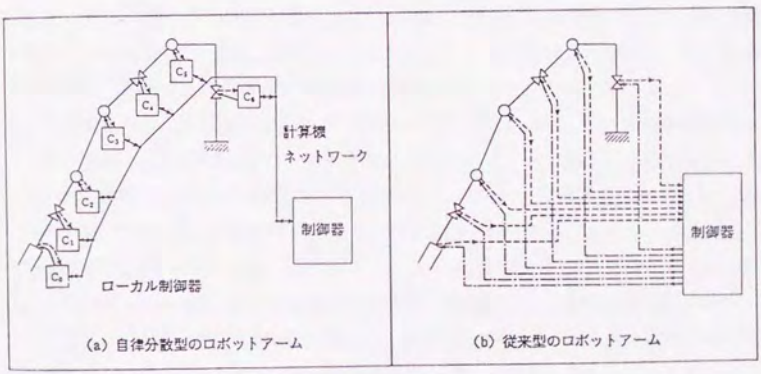


Figure 6.57: 分散マニピュレータの概念図

今、通常状態で上位制御計算機から肘上げの指令メッセージが与えられると、Figure 6.58のような経路で伝達されたメッセージは第3関節部で実行され、破線の経路で動作の完了を通知する Ack メッセージを返送する。ここで、モータの故障が検出されていると、その代替動作は Figure 6.59のようになる。つまり、第3関節に伝達された肘上げメッセージは、摑み上げのメッセージに変換されて送出され、指部で実行されたあと、更に手首に対する持ち上げメッセージに変換される。手首での実行が完了すると、初めてここから上位制御計算機に対して Ack が送信される。Figure 6.61は通常時、及び Figure 6.62は第3関節部モータ故障時のメッセージ交換の様子である。このメッセージ交換の様子は実際にトレースとして得られたものではないが、仮にメッセージ送信のトレースが記録できたとすると、その情報からはこの図と同等の図が得られるはずである。よって、ここでは何らかの手段によってトレース情報が得られたものとして議論をすすめる。

6.7.2 VOGUE による視覚化

一方、Figure 6.63は通常時、Figure 6.64は第3関節モーター故障時のメッセージ交換図に相当するものを3次元上に記述したものである。ノードの配置はグラフの見易さに大きく影響するが、これには適用領域ごとの文化、ノウハウなどがあり一概に議論することは不可能である。本例ではメッセージが見易いように適当に配置を決定している。上位制御計算機からのメッセージは常にネットワーク・マネージメント・プロセッサ (以後 NMP と略記) を経由して各関節に伝送されるので、上位制御計算機に対応するノードを画面左、NMP に対応するノードを中央付近に配置し、他の計算機に相当するノードをこの周囲に円弧状に配置することで、策走するメッセージの視認性を上げている。画面上から右にかけて各々第4、第3、第2、第1関節に対応するノード、画面下方が指部、その左がセンサに対応するノードである。z 軸方向から見れば計算機間のメッセージ通信の様子を表現し、これに対して垂直な方向から見れば、Figure 6.61、Figure 6.62と同等のメッセージ交換図が得られる。ただし、2次元上に実現された Figure 6.61、Figure 6.62 のメッセージ交換図の計算機に対応する軸の順番は、マニピュレータにおける計算機接続順序を考慮し、上位制御計算機、NMP、第4関節部、第3関節部、第2関節部、第1関節部、指部、センサ部の順序で並べられている。VOGUE でこれと同等の図を得るならば、相応の配置を工夫すれば十分実現可能である。

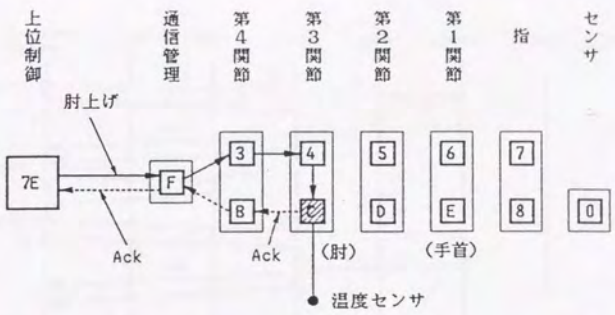


Figure 6.58: 正常時のメッセージ伝達経路

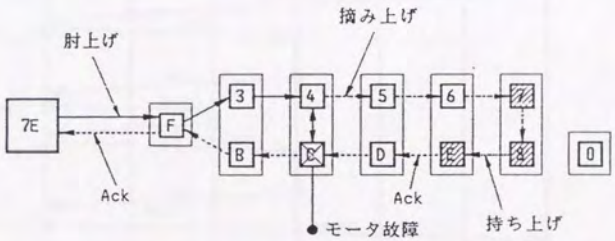


Figure 6.59: 第 3 関節モーター故障時のメッセージ伝達経路

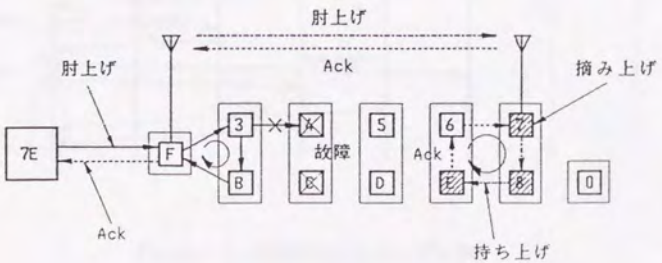


Figure 6.60: 第 3 関節計算機故障時のメッセージ伝達経路

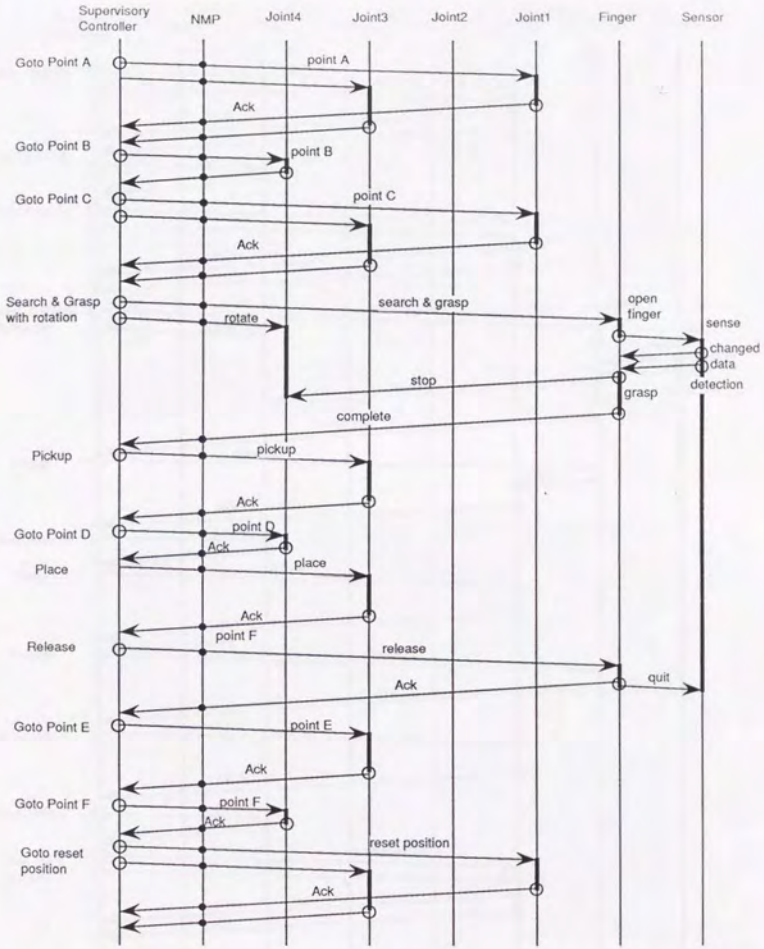


Figure 6.61: 正常時のメッセージ交換

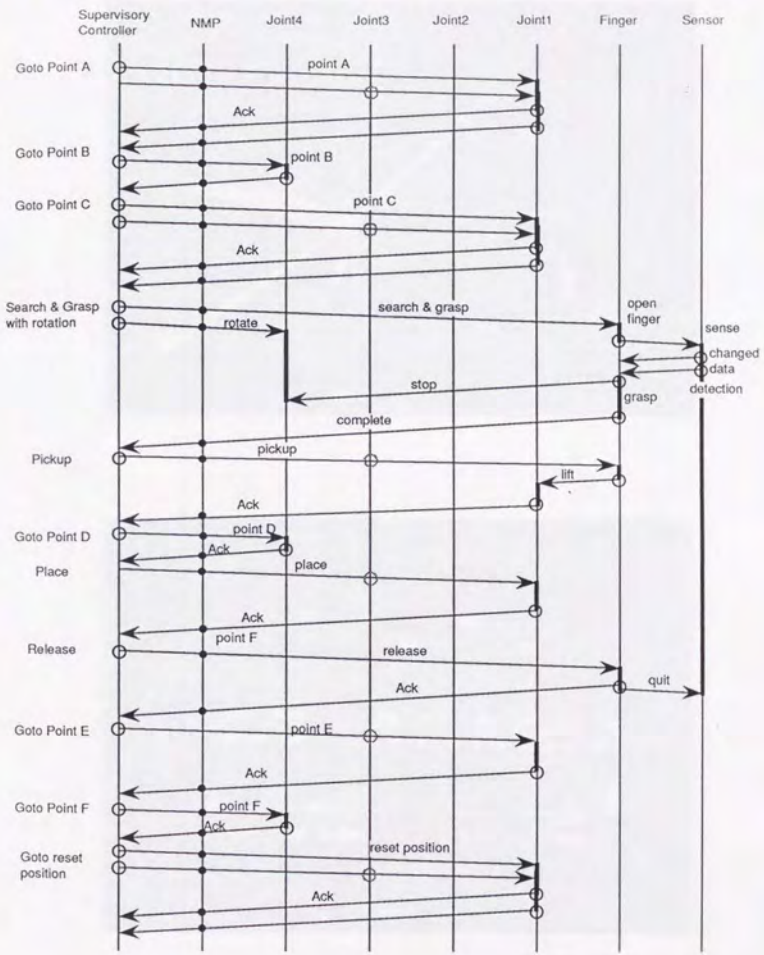


Figure 6.62: 第 3 関節モーター故障時のメッセージ交換

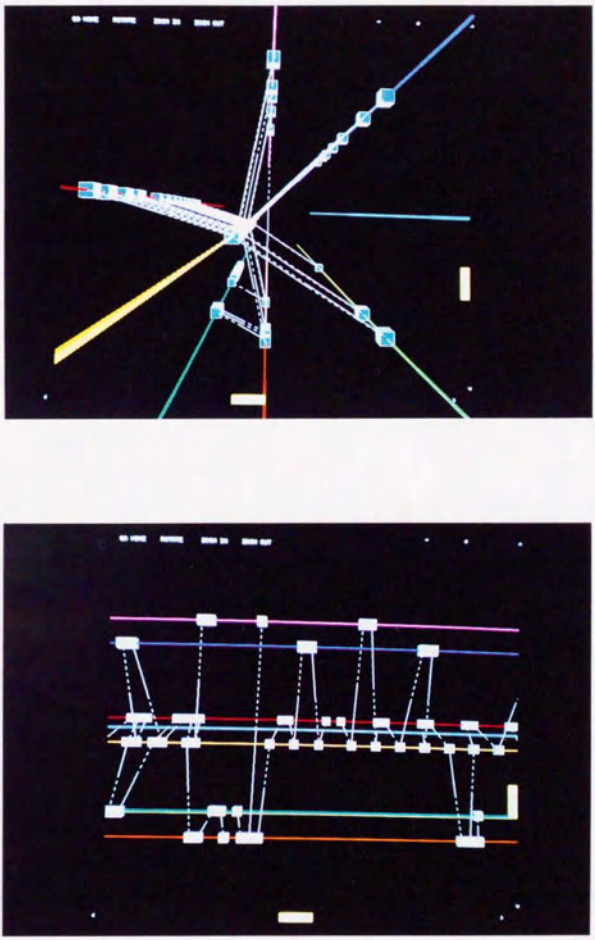
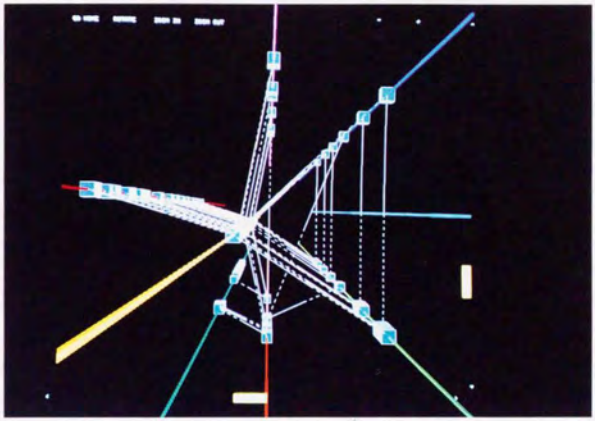


Figure 6.63: 正常時のメッセージ交換の視覚化



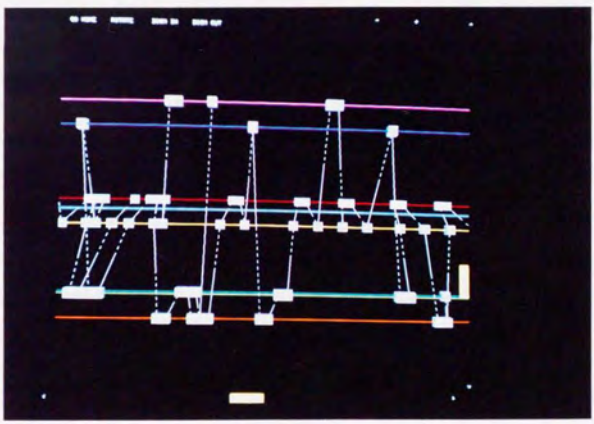


Figure 6.64: 第 3 関節モーター故障時のメッセージ交換の視覚化

さて、第 3 関節部モータが故障した場合、第 3 関節に送られたメッセージは肘上げメッセージとして指部に伝達される。平面的メッセージ交換図においてはこれが矢印として表現されている。その時、メッセージが素通りした部分は単に矢印が関節を表現する線を横切るだけであるが、経由した計算機の部分は黒い丸で示されている。注意して見ないと、上位制御計算機からのメッセージは NMP を経由したのかどうか判りにくい。通常、上位制御計算機、NMP、第 3 関節部と伝達されるメッセージが、第 3 関節モータ部が故障している場合には、上位制御計算機、NMP、第 3 関節部、第 1 関節部と伝送されるが、経由した計算機が判りにくい。

これに対して VOGUE に実現されたトレースの場合、伝送メッセージはそれぞれの計算機に対応するノードを経由する折れ線として表現されるため、上で述べたような誤認をする可能性は非常に低くなる。これは単に 3 次元グラフの 2 次元グラフに比較した視認性の高さという問題ではなく、計算機の平面的広がり情報と時間情報の 2 つが同時に認識可能だという事実によるものであると考えられる。

ここで Figure 6.61のような 2 次元の図においても、例えば第 1 関節部と第 3 関節部に相当する線を入れ換えることによって、メッセージ伝送を示す矢印を直線でなくし、第 3 関節経由の事実をより判り易く表示することは可能であるとの反論があるかもしれないが、それは本例における上位計算機、NMP、第 3 関節部、第 1 関節部という特殊なケースにおいてのみ有効な方法であって、一般のメッセージ伝達には対応しきれない。一方、3 次元図の場合にはもちろん初期の配置にもよるが、一般的に言ってこれらのメッセージ伝達順番を誤認する可能性は低い。これが 3 次元図によって計算機間通信と時間軸を同時に見る利点である。

もう一つの例として、マニピュレータの第 3 関節部が計算機ごと故障している場合について考えてみる。計算機が故障した場合には、主要通信路が使用不能となるため、補助的な通信路が使用される。故障した計算機が同定されると、無線通信チャンネルによって指部へメッセージを送信する。Figure 6.60はこの時の肘上げメッセージに対するメッセージの流れ、Figure 6.65はタイミングチャートである。NMP から指部へのメッセージは矢印によって示されているが、この場合にもやはり矢印は各関節部の線と交差しなければならず、直接指部へ送られたメッセージか、関節のどれかを経由してきたものかを認識するのに労力を要する。これに対して Figure 6.66は VOGUE で記述されたものであるが、NMP から指部へのメッセージは直線的に表示されており、直接送られてきたものあることが即座に理解できる。

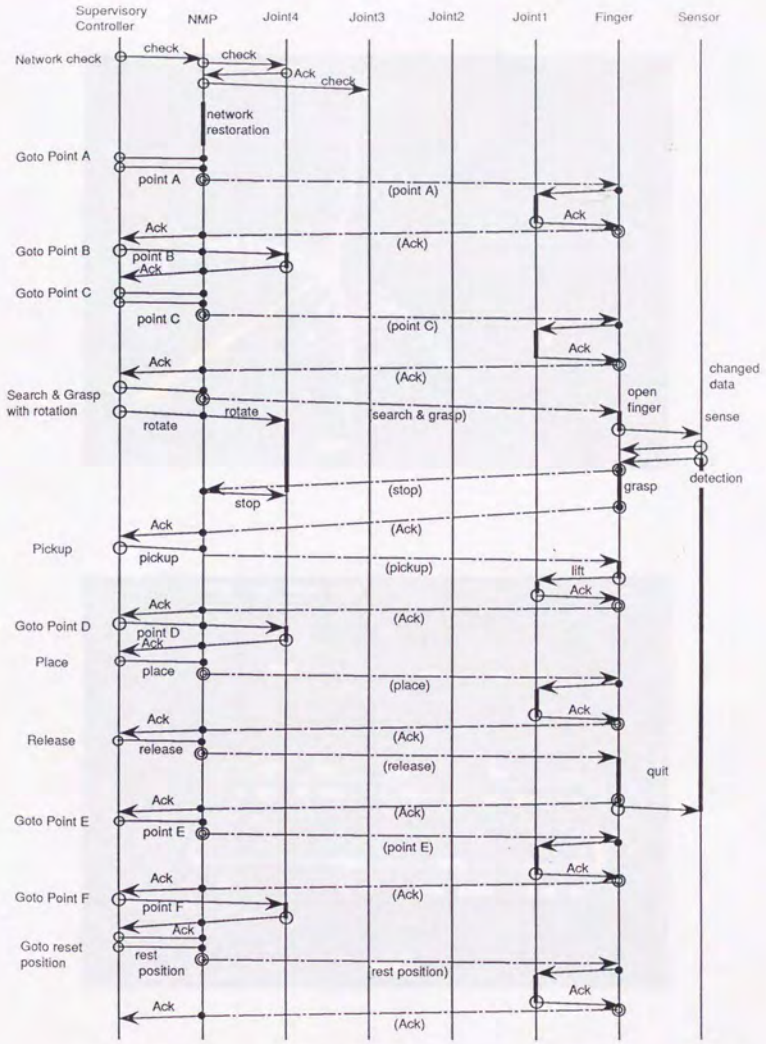


Figure 6.65: 第 3 関節計算機故障時のメッセージ交換

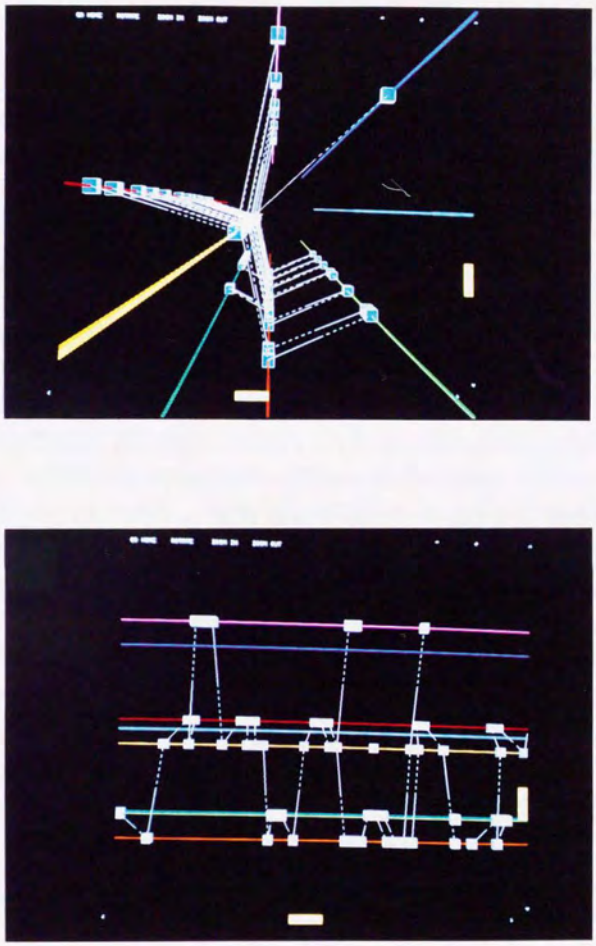


Figure 6.66: 第 3 関節モーター故障時のメッセージ交換の視覚化

また正常時と比較した場合、最初のネットワーク・チェックを除くと、NMP と第 3 関節部との通信が皆無である点、同様に NMP と第 1 関節部との通信が皆無である点、第 1 間節部と指部との通信が増えている点、指部から第 4 関節部への通信が消えている点等が、視覚的パターンとして容易に検出できる。

次にデバッグという側面から 3 次元視覚化について考えてみる。マニピュレータを動作させている時に、何らかの障害が起きマニピュレータが誤動作をした場合を想定すると、この段階では障害の原因は検出できていない。仮にトレースデータが残されていたとして、これを視覚化することによって図のような表示が得られたとすると、通常状態では第 3 関節部を経由して第 1 関節部に向かうメッセージは存在しないことは事前に判っているため、断定はできないがこのメッセージに関連する部分に何らかの不都合が生じているという可能性が生じる。特に第 3 関節部で処理されるべきメッセージが、第 1 関節部に伝達されている点を重視すれば、第 3 関節部のハードウェアあるいはソフトウェアに不都合が生じているとの仮説が立てられる。同様に図のような視覚化が得られた場合には、通常存在しえない NMP から指部への直接メッセージがあることが認識でき、これにより各計算機のいずれかが故障しているのではないかとの推測も成り立つ訳である。当然、障害には様々なケースがあり、絶対と言えるわけではないが、全く情報がない状態に比較しその障害を検知する確率ははるかに高くなると考えられる。また、このような図からの障害箇所の推定は 2 次元図でも可能であるが、前述したとおり 2 次元の場合、計算機間のメッセージ伝達の様子が 3 次元に比べ、把握しにくいという欠点がある。

以上まとめると、並列計算機システムにおけるメッセージ伝達は、時間に関する視点と計算機間の平面的広がりという視点を 3 次元を使って同時に見ることによって、2 次元に比べ明らかに見易くなる。

6.8 本章のまとめ

本章では、電力制御用ソフトウェアを取り上げ、そのモジュール構造とタスク間通信の VOGUE による視覚化、及び時間軸を導入して拡張したプロセス・モニタの実現について述べた。電力制御用ソフトウェアのタスク間通信は、多くのタスクと階層的に整理できる多くの種類のタスク起動方式が存在し、かつこれらが複雑に関連しあったネットワーク構造を形成している点において、VOGUE の図形数制御手法の例題として適切なものであると考える。さらに、タスク間通信とその時間的関係という 2 つの視点を有する点において、VOGUE による 3 次元視覚化の例としても適切なものであると考える。

モジュール構造とタスク間通信の視覚化においては、多くのタスクをスケール変換を用いた領域図として表現することで、モジュール階層関係とタスク間通信関係を同時に表示することができた。タスク間通信はその起動タイプによりオブジェクト指向 DB を用いて階層的に管理されるため、クラス階層での焦点の移動によってリンクの表示数を制御でき、着目する起動方式によって必要な情報のみが表示できた。また、第 4 章で数値的に示された Fractal Views の図形数制御能力が、実際のネットワーク構造においても検証された。さらに実際の電力制御用計算機のプロセスの実行状態を記録したトレース情報を用いて、プロセス・モニタへと拡張できることを示した。作成したプロセス・モニタは他のシステムへ十分適用可能な程の一般性を有し、プロセスの実行状態が OS レベルでサポートできるならば意味のある視覚化表現を得ることができる。

作成されたプロセス・モニタを用いて、実際の電力制御用計算機における障害事例の視覚化を行ない、従来はトレース・データからボトムアップ的に行なわれていたデバッグ作業を、視覚化表現を用いてトップダウンに行なえる可能性について考察を行なった。一般にプロセスの実行状態は実行時の環境に依存し、同一条件で試験を行なっても微妙なゆらぎが存在する。つまり、トレース情報はその詳細においては毎回微妙に異なるため、ボトムアップ的手法によるバグの同定はかなりの労力を要する。一方、図形表現を用いることでこれらのゆらぎの吸収が可能で、図形全体としての形状からまず無限ループ、デッドロック等、障害のパターンを同定し、しかる後に詳細を調べることが可能となると思われる。

取得したトレース情報は 1CPU のものであったが、将来的には並列分散システム全

体の視覚化を考える必要がある。そこで、小規模ながら並列分散システムとしての本質を有する分散マニピュレータの作業時における各計算機間のメッセージ通信の様子を視覚化した。通常状態、関節部モーター故障時、関節部計算機故障時の各場合を視覚化すると、2 次元的には把握の困難な情報が 3 次元的な視覚表現からは容易に得られることを示した。この結果は将来的大規模並列分散システムの開発にも適用できるものと考える。

第 7 章

適用例 2: オブジェクト指向言語のクラス・ライブラリ

7.1 緒言

本章では、VOGUE による 3 次元ソフトウェア視覚化の例として、オブジェクト指向言語のクラス・ライブラリの視覚化を取り上げる。クラス階層はオブジェクト指向言語の特徴の 1 つであるが、プログラミング、及びデバッグの際のメソッド探索には、実際に起動されるメソッドがどれかを意識している必要がある。つまり、クラス階層関係とともにメソッド継承関係をも意識しつつ開発を行なわねばならない。この 2 視点は従来、別々の異なる図として表現されていたが、第 3 章で述べたように 1 つのモデルに対し 2 つの図形を与えられたユーザのメンタル・モデルに対する悪影響は避けられない。一方、この 2 視点は VOGUE を用いることで 1 つの図として表現することが可能であり、VOGUE の 3 次元機能の例題として適当なものであると考えられる。

以下、まず 7.2 節においてオブジェクト指向言語のメソッド探索における視点について述べる。次に 7.3 節において VOGUE によるクラス・ライブラリ視覚化の手法について述べた後、7.4 節において従来ツールとの比較を目的として行なった被験者実験に関して述べる。さらに 7.5 節ではより複雑な例への対応として Flavor のデーモン・メソッドへの対応と評価実験に関して述べ、7.6 節では実際の Smalltalk-80 のクラス Collection 以下のクラス・ライブラリの視覚化を行ない、大規模データ視覚化における問題点を議論する。最後に 7.7 節において考察を行なう。

7.2 メソッド探索における 2 視点

第 3 章で述べたように、オブジェクト指向言語においてはインスタンス間に様々な関係が存在する。本章ではこの内特にメソッド探索という側面から、考察を行なう。

オブジェクト指向言語のプログラミングは、ユーザが新たなクラスを作成するか、あるいは既存のクラス・ライブラリ中から要求に近いクラスを探して、そのサブクラスを定義することによって行なわれるが (差分プログラミング)、こうしたオブジェクト指向におけるメソッド探索においては以下の主要な 2 つの視点が存在する。

- **クラス階層に着目した視点**
  インスタンスの内部変数 (Smalltalk-80 のインスタンス変数、Flavor のスロット等) は、スーパークラスのそれを継承する。よってプログラマは常に着目しているクラスより上位のクラスを意識する必要がある。

- **メソッド継承に着目した視点**
  メソッドは基本的にスーパークラスのものが継承されるが、必要に応じて下位のクラスで再定義される。よって、あるインスタンスへ送られたメッセージがどこで処理されるかを意識する必要がある。

オブジェクト指向言語におけるメソッド継承は、開発時にはコード再利用という点から非常に便利な機能であり、ソフトウェア危機に対する手段としてオブジェクト指向言語が注目を浴びている主たる要因の一つである。一方で、あるインスタンスへ送られたメッセージは必ずしもそのインスタンスが属するクラスに存在するとは限らない。そのインスタンス直属のクラスにメソッドが存在しない場合、クラス階層を下から順番にたどっていき、最初に現われたメソッドが起動されることになる。

こうしたメソッド継承の複雑なメカニズムは、実行時にはシステムによって自動的に行なわれるため意識する必要はあまりないが、デバッグ時にはプログラマが一つ一つ追っていく必要がある。実際、Smalltalk-80 におけるシステム・ブラウザでは、対応するクラスのメソッドを探し、存在しなければクラス定義に戻ってその直接のスーパークラスを調べ、そこでまた同様の手続きを繰り返すことになる。このプロセスをフローチャートで表すと Figure 7.1のようになると思われる。

こうした手続きの煩雑さは、Smalltalk-80 の標準システム・ブラウザが、クラス階層を視覚的にサポートしていないことに起因すると思われる。このシステム・ブラウ

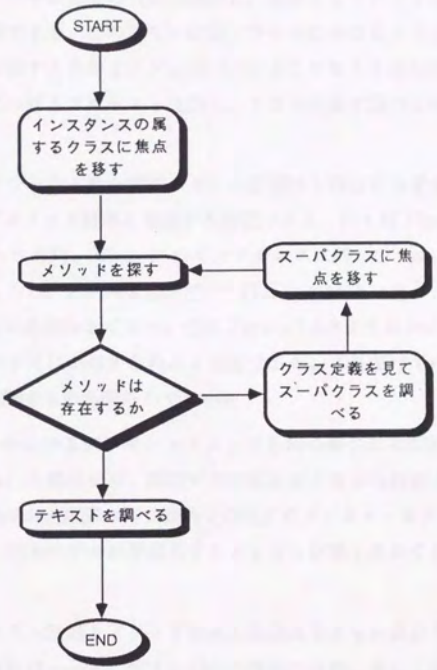


Figure 7.1: Smalltalk-80 のシステム・ブラウザでのメソッド探索

ザの欠点を補うものとしては、例えばクラス階層ブラウザ [8] がある。これはクラス階層図を木構造として提示し、クラスに対応するノードをクリックすることで属するメソッドがポップアップ・メニューとして提示される。さらにメニューを選択することで対応するソースコードにアクセスすることが可能である。クラス階層が常時視覚的に支援されることでプログラマの記憶の負荷は減少し、結果としてソフトウェアの開発効率が高まることが期待できる。このクラス階層ブラウザにおけるメソッド探索プロセスをフローチャートで表すとおおよそ Figure 7.2のようになると思われる。先の標準システム・ブラウザにおけるプロセスと比較し、クラス定義を調べる部分が短縮されるのがわかる。

最初に述べたように、オブジェクト指向言語のクラス階層は 1 視点に過ぎず、実際に開発を行なう場合には必ずメソッド継承を考慮する必要がある。例えば Figure 7.3のようなクラス階層図が与えられた時、Class-5 のインスタンスに対して test というメッセージを送ることを考えると、クラス階層図だけではこの test というメソッドが実際にはどのクラスのものなのか明らかでない。逆に Figure 7.4のような test というメソッドを持つクラスのリストだけが与えられたと仮定すると、これだけでは、どのクラスに属するメソッドが起動するのか明らかでない。

後述する Flavor のようないわゆるデーモン・メソッドを持つ場合には問題がさらに複雑になる。両方が与えられた場合には、両図での対応を取りながら起動されるメソッドを同定することは可能だが、両図の持つ制約を満足するメンタル・モデルをユーザが形成する負荷は大きく、例えモデルが形成できたとしても記憶しておくのは不可能に近い。

以上のような理由から、クラス階層とメソッド継承の両視点をともに満足するような視覚表現が可能ならば、それはユーザに対する負担の軽減のため、そして最終的にはソフトウェア開発効率に対して良い影響を及ぼすものと考えられるが、本論文が主張する 3 次元視覚化にはそれが可能である。

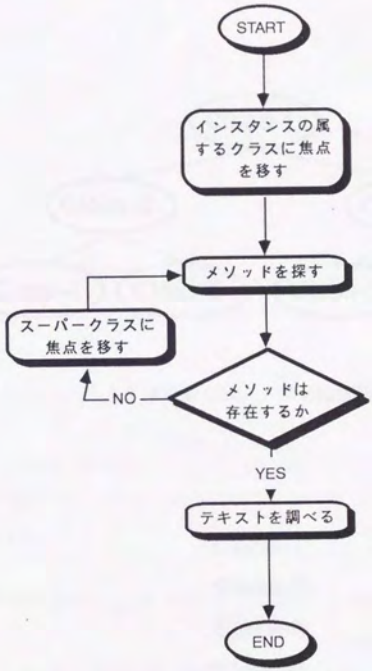


Figure 7.2: Smalltalk-80 のクラス階層ブラウザでのメソッド探索

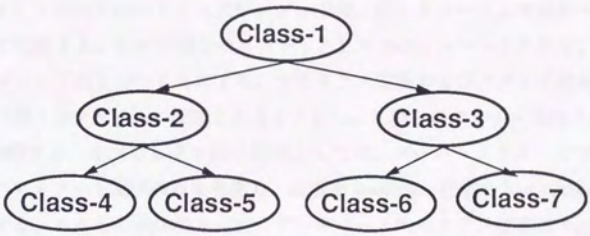


Figure 7.3: クラス階層図

Class-1
Class-2
Class-4
Class-6

Figure 7.4: メソッド "test" を持つクラスのリスト

7.3 VOGUE によるクラス・ライブラリの視覚化

前節では、クラス階層とメソッド継承の両視点を共に満足する視覚表現が必要であることを述べた。本節では、VOGUE によるこの視覚表現の実現方式について述べる。

7.3.1 対象のモデル化

オブジェクト指向言語のクラス及びメソッドは、データベース上では各々一つのクラスとして定義する。各々が固有のスロットとして name スロットを持ち、クラス名あるいはメソッド名を持つことにする。またクラス定義およびメソッド定義は、便宜上 1 つ 1 つ別々のファイルに記述されることとし、これらのファイル名はスロット filename に保持する。オブジェクト間の関係としては、スーパークラス・サブクラス関係とクラス・メソッド関係のみを考慮し、前者を isa-link、後者を has-method-link として定義する。これらの具体的な定義とデータベース内のクラス構造は Figure 7.6のようになっている。

また実験のためにメソッドに対応するノードが選択された場合、各定義を参照できるようにする。ただしその時 Emacs 等のエディタを起動しファイルを調べることは可能であるが、後出の実験においては可能な限り定義参照の時間を短くしたい。よって、ノードが選択された場合、各定義は Lisp 上に表示することとし、これらは以下の定義によって実現される。

```
(defmethod left-button-proc ((obj oop-method)(g db-grapher)
                             &rest args)
  (with-open-file (st (make-pathname
                       :name (filename obj) :type "lisp"))
    (do ((line (read-line st nil nil)(read-line st nil nil)))
        ((null line))
        (format t "~%~A" line))))
```

```
(define-class oop-class ()
  ((name :initarg :name :accessor class-name)
   (filename :initarg :filename :accessor filename)) )
(define-class oop-method ()
  ((name :initarg :name :accessor method-name)
   (filename :initarg :filename :accessor filename)) )

(define-link isa-link () ())
(define-link has-method-link () ())
```

Figure 7.5: モデルの定義

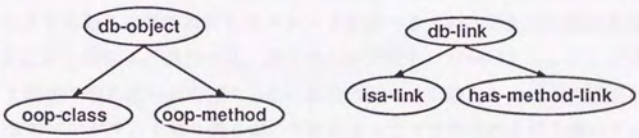


Figure 7.6: データベース内の構造

7.3.2 グラフへの描画

クラスとメソッドのインスタンスは各々ノードとして表示し、各関係はリンクとして表示する。クラスの階層構造は木構造として xy 平面に配置し、各メソッドはこの平面と垂直な方向、つまり z 軸方向にメソッドが所属するクラスに対応するノードと同一の x 座標及び y 座標を持ち、かつ同じ名前を持つメソッドは同一 z 座標を持つように配置する。この処理は Figure 7.7によってなされる。

Figure 7.8が実現されたクラス・ライブラリの視覚化である。xy 平面に垂直な方向から見るとクラス階層図、真横からみるとメソッド継承図を表し、前述した 2 視点をともに満足する表現が得られる。この場合におけるメソッド探索のフローチャートは Figure 7.9のようになると思われる。つまり、インスタンスに送られたあるメッセージに対して実際に起動されるメソッドが視覚的、かつ 1 つの図で把握できる。

7.3.3 ブラウザの一般性

ここで実現された視覚表現は Smalltalk-80、Flavor、C++ 等、メッセージ送信式の第 1 引数をメッセージのレシーバとする一般のオブジェクト指向言語には適用可能であるが、CLOS のようにメッセージのレシーバを特定せず、引数すべてを対等に扱う場合には適用不可能である。なぜならば、CLOS では同じ名前のメソッドは総称関数という概念によってまとめ、ある特定のクラスに所属するという従来のような捉え方をしない。いわばメッセージの受け手は引数全てであるとの立場をとるため、本ブラウザのようにあるクラスに対してメソッドを並べるという考え方は馴染まない。ただし、引数が 1 個以下の場合には、便宜上その引数をレシーバとみなすことによって Flavor と同様の取り扱いが可能である。以後の例では引数を 1 つ以下に限定することで CLOS のプログラムを取り扱うが、これによってブラウザの本質を損ねることはないと考える。

7.3.4 各インスタンスの生成

VOGUE では個々のインスタンスの登録は、アプリケーションごとにユーザによって各々行なわれる必要がある。本例では CLOS のファイルを解析し、クラスおよびメソッドに対応するオブジェクトをデータベース内に生成し、対応するリンクを張るパーザを作成し使用した。パーザは CLOS のファイルを読み、クラス定義を見つけると対

```
;; まずクラスを木構造配置
(layout-as-tree グラフ (assoc-node グラフ 'label ルートクラス名)
   :child-function
   #'(lambda (n)
       (mapcar #'link-destination
         (remove-if-not #'(lambda (l)
                            (typep (figure l) 'isa-link))
                        (output-links n)))))
;; メソッド名の表を作る
(setf method-name-table (make-hash-table :test #'equal))

;; z 座標の初期値とオフセット
(let ((z 10)
      (zoffset 20))
  ;; グラフの全ノードに対して
  (map-nodes (node グラフ)
    ;; obj は node の実体
    (let ((obj (figure node)))
      ;; oop-method の時だけ処理する
      (when (typep obj 'oop-method)
          ;; 表に同じ名前があればその z 座標、
          ;; なければ新しい z 座標
          (let* ((zpos (or (gethash (method-name obj)
                                    method-name-table)
                           (setf (gethash (method-name obj)
                                          method-name-table)
                                 (incf z zoffset))))
                 ;; has-method-link の始点がクラスである
                 (class (car (solve '(? x) 'has-method-link obj)))
                 ;; クラスのノードを求め xy 座標は同一値にする
                 (cnode (assoc-node グラフ 'figure class))
                 (xpos (xpos cnode))
                 (ypos (ypos cnode)))
            ;; 実際に動かす
            (change-attribute node :xpos xpos :ypos ypos :zpos zpos))))))
```

Figure 7.7: クラス・メソッドレイアウトの概要

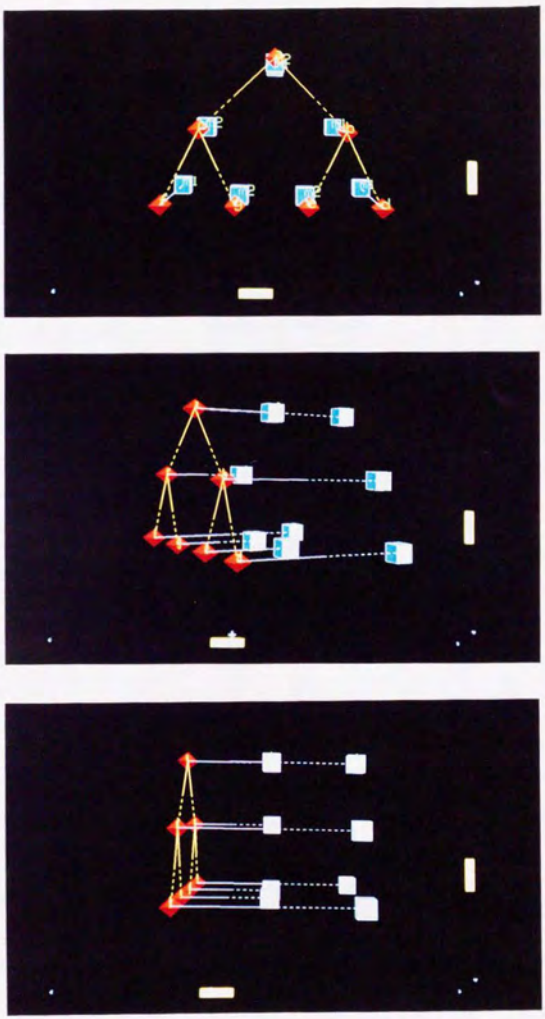


Figure 7.8: VOGUE で実現したクラス・ライブラリの視覚化

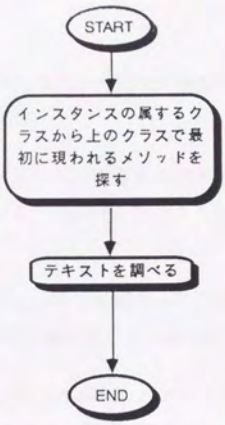


Figure 7.9: VOGUE でのメソッド探索

応するデータベース上にクラス・オブジェクトを生成し、上位クラスとリンクで接続する。メソッド定義を見つけると、メソッド・オブジェクトを生成し、所属するクラスとリンクで接続する。

7.4 被験者実験による有効性の検証

7.4.1 実験概要

VOGUE で実現された 3D 視覚化表現の有効性を検証するために、2 次元視覚化表現との比較実験を行なった。ただし、

1. 現在のウィンドウ・システム上に実現された既存のブラウザは、メニュー・ボタン等多くの機能を保持するため試作システムとの単純な比較は意味をなさない。

2. 図形の選択からテキストの表示までに要する時間等、2 次元と 3 次元の本質的な違いに無関係な点は、両者で差が生じないようにする必要がある。

という 2 点を考慮し、Smalltalk-80 上に実現されているクラス階層ブラウザの本質を損なわない範囲で、VOGUE 上にクラス階層ブラウザをシミュレートしたツールを作成した (Figure 7.10、Figure 7.11)。クラスの階層構造が xy 平面上の木として表現され、任意のクラスに対応するノードをマウス・クリックすると、そのクラスに属するメソッドが垂直下に出現する。さらにメソッドに対応するノードをクリックすると、メソッドの定義が SUN 側のウィンドウ上に表示される。別なクラスを選択するとそれまで表示されていたメソッドのノードは消え、新たなメソッドがポップアップする。これらは、やはり left-button-proc で実現されている。以後、これを 2D ブラウザと呼び、3 次元に実現されたものを 3D ブラウザと呼ぶことにする。

7.4.2 実験方法

実験は被験者 8 人を用いた個人実験で以下のように行なった。まず、簡単な例題でツールの使用法に関する説明を行ない、被験者に十分ツールになれてもらった後、3D ブラウザに Figure 7.12のクラス・メソッドの組合せを表示する。その状態で、Figure 7.13に示すメッセージ送信式をこの順番で 1 つずつ記述したカードを被験者に与える。被験者には、そのメッセージ送信式が最終的に返す値を次々に紙に書いてもらう。ここで、カードを用いたのは前出の答えを後出の問題で再利用しないよう配慮したためである。全部の答えを書き終ったら、Figure 7.14,Figure 7.16を同様に解いてもらった。さらに 1 日後、今度は 2D ブラウザで同様の実験を行なった。

Figure 7.10: VOGUE でシミュレートしたクラス階層ブラウザ (初期状態)

Figure 7.11: VOGUE でシミュレートしたクラス階層ブラウザ (メソッドのポップアップ時)

```
(defclass one () ())
(defmethod test ((obj one))
  1)
(defmethod result1 ((obj one))
  (test obj))

(defclass two (one) ())
(defmethod test ((obj two))
  2)

(defclass three (two) ())
(defmethod result2 ((obj three))
  (result1 obj))
(defmethod result3 ((obj three))
  (test obj))

(defclass four (three) ())
(defmethod test ((obj four))
  4)
```

Figure 7.12: 実験に用いたクラス階層 A

```
(test (make-instance 'one))
(result1 (make-instance 'one))
(test (make-instance 'two))
(result1 (make-instance 'two))
(test (make-instance 'tree))
(result2 (make-instance 'three))
(result3 (make-instance 'three))
(result1 (make-instance 'four))
(result2 (make-instance 'four))
(result3 (make-instance 'four))
```

Figure 7.13: 実験に用いたタスク A

```
(defclass one () ())
(defmethod test ((obj one))
  1)
(defmethod foo ((obj one))
  (test obj))
(defmethod bar ((obj one))
  (foo obj))

(defclass two (one) ())
(defmethod test ((obj two))
  2)

(defclass three (one) ())
(defmethod bar ((obj three))
  (test obj))
(defmethod foo ((obj three))
  (let ((instance (make-instance 'two)))
    (test instance)))
```

Figure 7.14: 実験に用いたクラス B

```
(test (make-instance 'one))
(test (make-instance 'two))
(test (make-instance 'three))
(foo (make-instance 'one))
(foo (make-instance 'two))
(foo (make-instance 'three))
(bar (make-instance 'one))
(bar (make-instance 'two))
(bar (make-instance 'three))
```

Figure 7.15: 実験に用いたタスク B

```
(defclass class1 () ())
(defmethod test ((obj class1))
  1)
(defmethod foo ((obj class1))
  (test obj))

(defclass class2 (class1) ())
(defmethod bar ((obj class2))
  (test obj))
(defmethod test ((obj class2))
  2)

(defclass class3 (class1) ())
(defmethod bar ((obj class3))
  (let ((instance (make-instance 'class4)))
    (test instance)))

(defclass class4 (class2) ())
(defmethod test ((obj class4))
  3)

(defclass class5 (class2) ())
(defmethod foo ((obj class5))
  (bar obj))

(defclass class6 (class3) ())
(defmethod test ((obj class6))
  4)
(defmethod bar ((obj class6))
  (foo obj))

(defclass class7 (class3) ())
```

Figure 7.16: 実験に用いたクラス C

```
(test (make-instance 'class3))
(test (make-instance 'class5))
(test (make-instance 'class7))
(foo (make-instance 'class1))
(foo (make-instance 'class2))
(foo (make-instance 'class4))
(foo (make-instance 'class5))
(foo (make-instance 'class6))
(bar (make-instance 'class2))
(bar (make-instance 'class3))
(bar (make-instance 'class4))
(bar (make-instance 'class6))
(bar (make-instance 'class7))
```

Figure 7.17: 実験に用いたタスク C

本実験では 2 次元と 3 次元の比較を目的とするために、ウィンドウ・システム上に実現されたツールに相当するものを作成し比較を行なったが、現在のウィンドウ・システムの段階で実現されうる従来形式のブラウザ上で、同様のタスクを完遂するのに要するおおよその時間を把握しておくことは意味のあることだと思われる。そこで厳密な比較を目的とするものではなく、あくまでも参考のために以下のような実験を行なった。

実現を容易にするため Smalltalk-80 のブラウザを借用し、Figure 7.12、Figure 7.14、Figure 7.16の CLOS のクラス関係、メソッド関係を定義した。クラスのリストから任意のものを選択すると、対応する定義と所属するメソッドのリストが出現し、さらにメソッドを選択するとメソッドの定義にアクセスできるという標準システムブラウザの機能が重要であり、プログラム自体が Smalltalk-80 か CLOS か、といった点は今の場合本質的でない。よって、なるべく前出の実験と条件が一致するように、コードには CLOS のものをそのまま用いた。実際には定義本体は Smalltalk-80 のコメントの形で記述され、インスタンス変数、クラス変数等、不必要な部分は極力削除した。メソッド・カテゴリとしては private のみを作成し、全てのメソッドがこれに属することとした。

7.4.3 実験結果

各被験者の課題達成までに要した時間のグラフをそれぞれ Figure 7.18から Figure 7.25に示す。また、各課題ごとに最短と最長のデータを除外した 6 データの平均をグラフにしたのが Figure 7.26である。すべての課題に対して 3 次元ブラウザの方が成績が良いのが見て取れる。

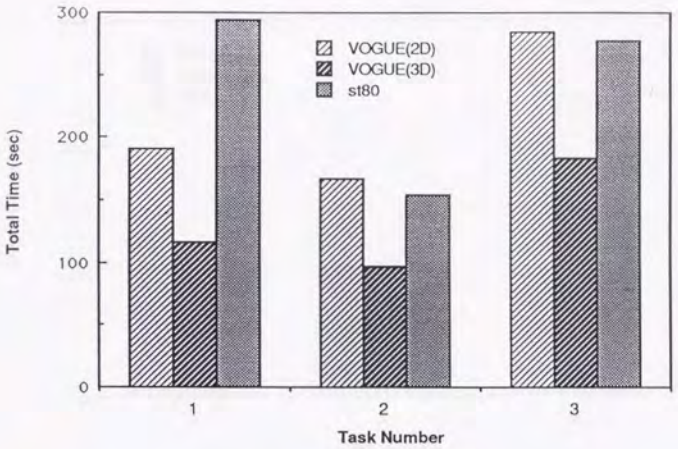


Figure 7.18: 実験結果 (被験者 A)

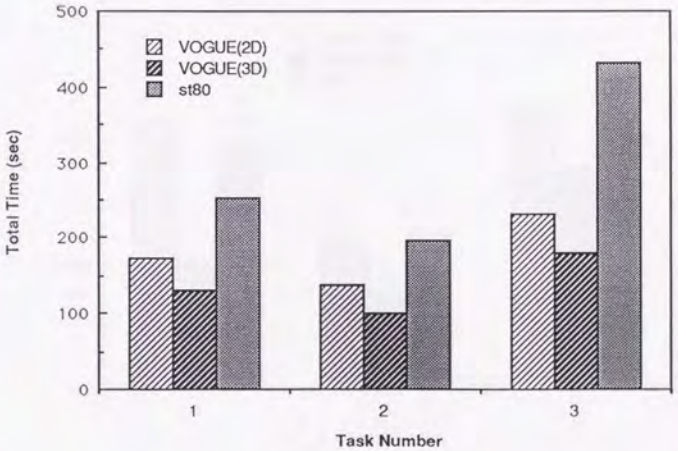


Figure 7.19: 実験結果 (被験者 B)

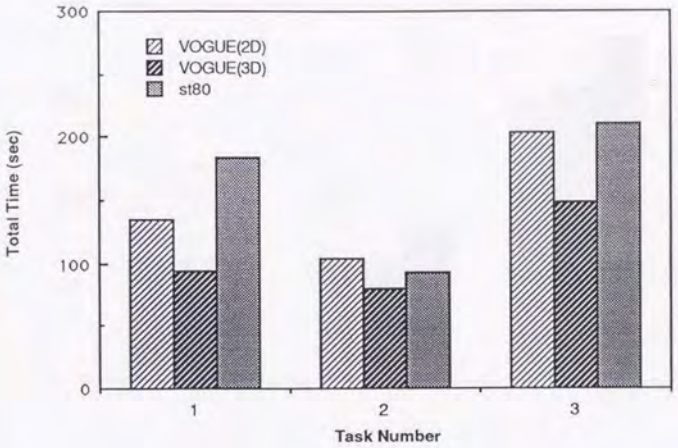


Figure 7.20: 実験結果 (被験者 C)

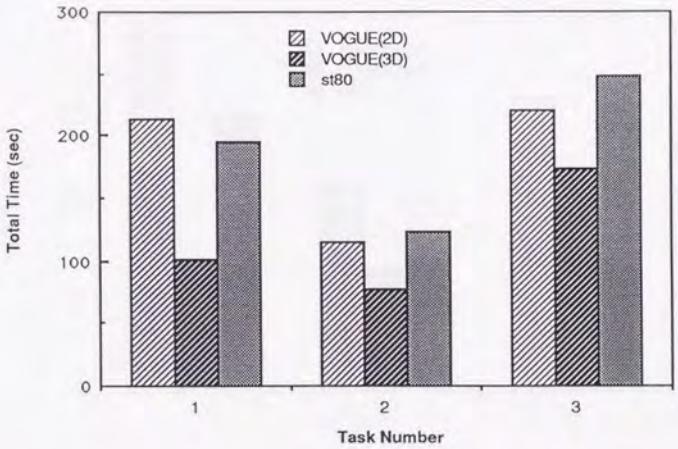


Figure 7.21: 実験結果 (被験者 D)

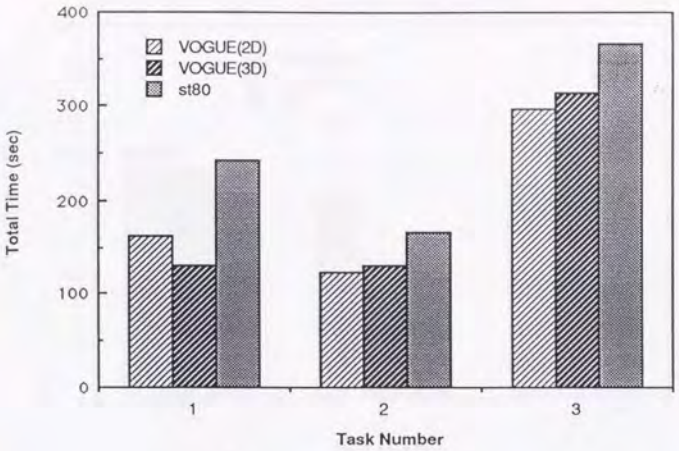


Figure 7.22: 実験結果 (被験者 E)

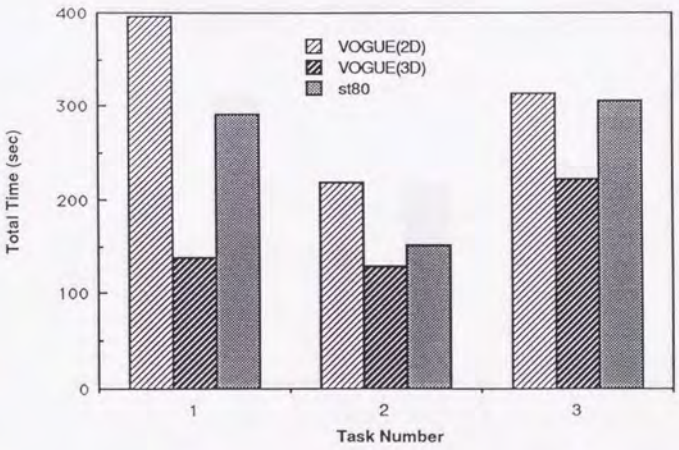


Figure 7.23: 実験結果 (被験者 F)

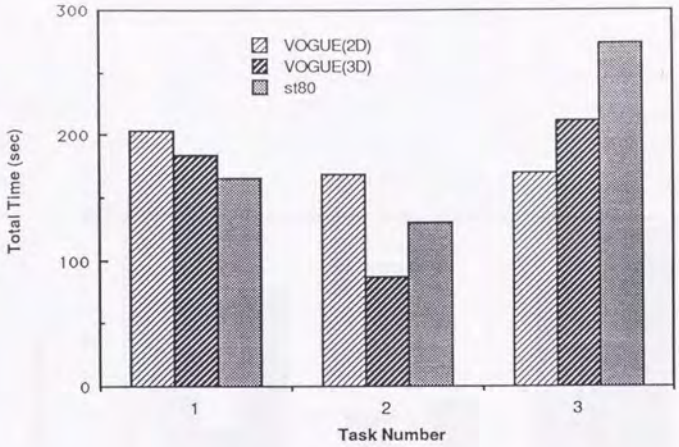


Figure 7.24: 実験結果 (被験者 G)

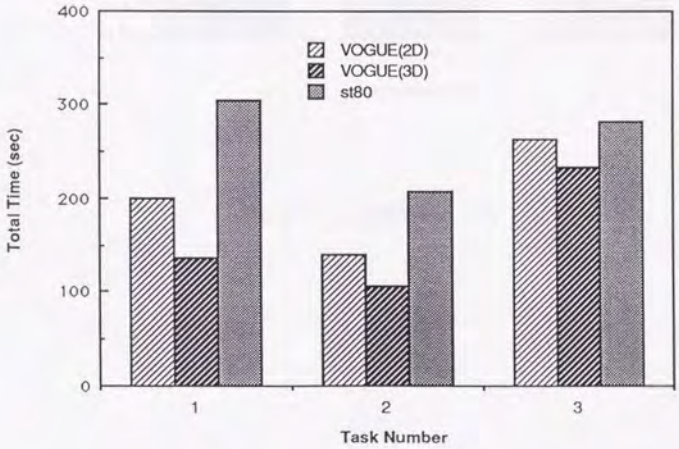


Figure 7.25: 実験結果 (被験者 H)

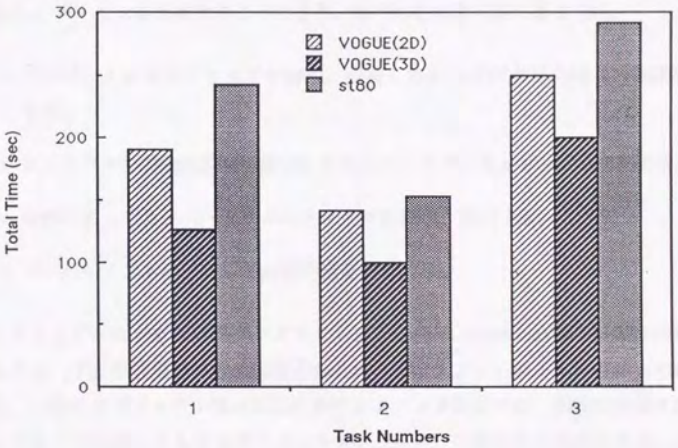


Figure 7.26: 実験結果の平均

7.5 より複雑な例での実験 -Flavor への対応 -

7.5.1 デーモン・メソッド

Flavor 系のオブジェクト指向言語にはデーモン・メソッドと呼ばれる特殊なメソッドが存在する。before メソッドは普通のメソッド (primary メソッド) が起動される前に、最も特定的なものから一般的なものの順序で起動され、 after メソッドは普通のメソッドが起動された後に、逆に最も一般的なものから特定的なものの順番で起動される。デーモンを利用することにより、以下の処理が可能となる [50]。

- スーパークラスのメソッドを継承、共有しながらサブクラス特有の処理を付加する。
- サブクラスで特別の環境を設定してスーパークラスのメソッドを実行できる。
- 条件によってスーパークラスのメソッドを継承、実行する。
- スーパークラスのメソッドが返す値を加工する。

こうしたデーモンを用いたプログラミング (method combination) は効果的に使用すればコードの再利用がさらに促進されるが、逆にメソッドのトレースは更に複雑になる。一般のオブジェクト指向言語におけるメソッド探索では、実際に起動されるメソッドを 1 つ特定しさえすればよかったが、Flavor の場合には起動される primary メソッドは 1 つであるが、関連する before、after の各デーモン・メソッドはすべて起動されるためこれらを全てトレースする必要がある。よってこの複雑な起動順番を標準システム・ブラウザで対応することはもはや不可能だと思われる。

これに対し現在行なわれている、デーモンの存在するメソッド探索の視覚的支援は Figure 7.27に示す方法である。各メソッドを起動順番に従って上から下へと、あるときには段付をした表形式で記述される。この表記の仕方は文献 [50, 58] でも用いられている方法である。

```
クラス A の :before メソッド
  クラス B の :before メソッド
    クラス C の primary メソッド
  クラス B の :after メソッド
クラス A の :after メソッド
```

Figure 7.27: 起動順番を示す表

しかし、この方法には次のような問題点がある。例えば Figure 7.28のように階層構造をなす 7 個のクラスに対し 2 種類計 16 個のメソッドが定義されているとする。

今、クラス d のインスタンスにメッセージ m1 を送った場合に起動されるメソッドは以下の表で表される。

```
クラス d の before メソッド
  クラス b の before メソッド
    クラス a の基本メソッド
  クラス b の after メソッド
クラス d の after メソッド
```

一方、クラス f のインスタンスにメッセージ m1 を送った場合に起動されるメソッドは次の表のようになる。

```
  クラス f の基本メソッド
クラス f の after メソッド
```

着目するインスタンスのクラスによって、同一メソッドでもその起動順番は大きく変化しその都度表を作成しなければならない。当然システムがこの起動順番のリストを作成し提示することは可能であるが、クラス階層図とこの動的に変化する表による支援だけでは、現在着目しているメソッドがこのクラス階層に対してどの程度定義されているのかはわからない。

クラス階層におけるメソッドの存在位置に対する理解不足は、既に定義済みのクラスのサブクラスを作成し、メソッドを定義する場合に特に問題となる。デーモンの存在しないオブジェクト指向言語の場合には、スーパークラスのメソッドを再利用する以外は新たなメソッドを定義すれば、既存の定義を全てキャンセルすることができた

```
(defclass a () ())
(defclass b (a) ())
(defclass c (a) ())
(defclass d (b) ())
(defclass e (b) ())
(defclass f (c) ())
(defclass g (c) ())

(defmethod m1 ((a a)) )
(defmethod m1 ((b b)) )
(defmethod m1 :after ((b b)) )
(defmethod m1 :before ((b b)) )
(defmethod m1 ((d d)) )
(defmethod m1 :after ((d d)) )
(defmethod m1 :before ((d d)) )
(defmethod m1 ((f f)) )
(defmethod m1 :after ((f f)) )

(defmethod m2 ((a a)) )
(defmethod m2 ((c c)) )
(defmethod m2 :after ((c c)) )
(defmethod m2 ((g g)) )
(defmethod m2 :before ((g g)) )
(defmethod m2 ((e e)) )
(defmethod m2 :before ((e e)) )
```

Figure 7.28: デーモンを持つクラス階層の例

```
(define-class primary-method (oop-method) ())
(define-class before-method (oop-method) ())
(define-class after-method (oop-method) ())
```

Figure 7.29: Flavor に対応したモデル定義

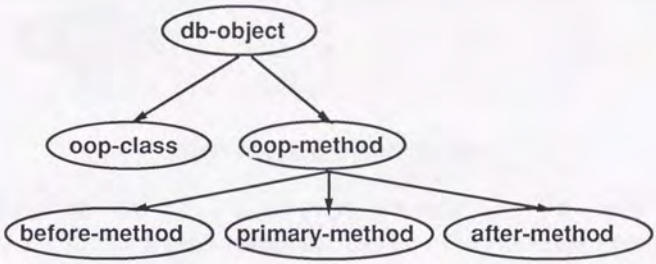


Figure 7.30: Flavor に対応したクラス階層

が、デーモンが存在する場合には新しく作成したクラスに対してメソッドを定義しても、基本的にスーパークラスに存在するデーモンはやはり起動される[1]。

以上の考察から、デーモンの存在するオブジェクト指向言語においては特に、クラス階層とメソッド継承の視覚的同時支援が重要であると思われる。VOGUE では 7.3 節で行なったクラス・ライブラリの視覚化に若干の変更を加えることで、こうした要求に対処することが可能である。次節ではモデルの変更点及びその視覚化について述べる。

7.5.2 VOGUE での対応

7.3 節での定義に Figure 7.29の定義を追加し、普通のメソッドは primary-method のインスタンスとする。これによってデータベース内のクラス階層は Figure 7.30のようになる。

[1] メソッド組合せを用いてユーザが起動するデーモンを制御することも可能であるが、メソッドの位置を理解していなければならない事実は変わらない

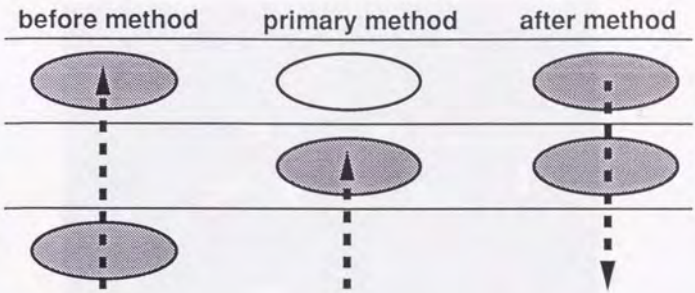


Figure 7.31: 起動順番の視覚的理解

グラフ上では before メソッド、after メソッドに対応するノードを、z 軸上で primary メソッドの各々前と後ろに配置する。Figure 7.32がこうして実現されたブラウザである。

実現されたブラウザは前節で述べた表記述のように、メソッド起動順番を上から下への直線的パターンで表現することはできないが、着目するクラスに対して before メソッドは下から上へ、基本メソッドは下から探索して最も最初に現れたもの、after メソッドは上から下へという視覚的パターンとして理解することができる (Figure 7.31)。このブラウザによって、新たなクラスを定義する場合に、対応するクラスから上に存在するデーモンを視覚的に把握してからメソッドを定義することができるため、デーモンに起因する意図しない副作用を極力さけることができるようになると考えられる。

7.5.3 被験者実験

7.4 節で行なったのと同様の比較実験を被験者 5 人に対して行なった。与えたタスクを Figure 7.33にしめす。尚、今回は Smalltalk-80 のシステム・ブラウザを用いた参考実験は行なわなかった。実験結果をグラフにしたのが Figure 7.35である。縦軸はタスク完遂に要した時間 (秒)、横軸は被験者を表す。このうち成績の最も良かったものと最も悪かった 2 つを除いて、平均を求めたのが Figure 7.36である。結果としてやはり 3 次元の方がメソッド探索は速やかに行なわれることが実証された。

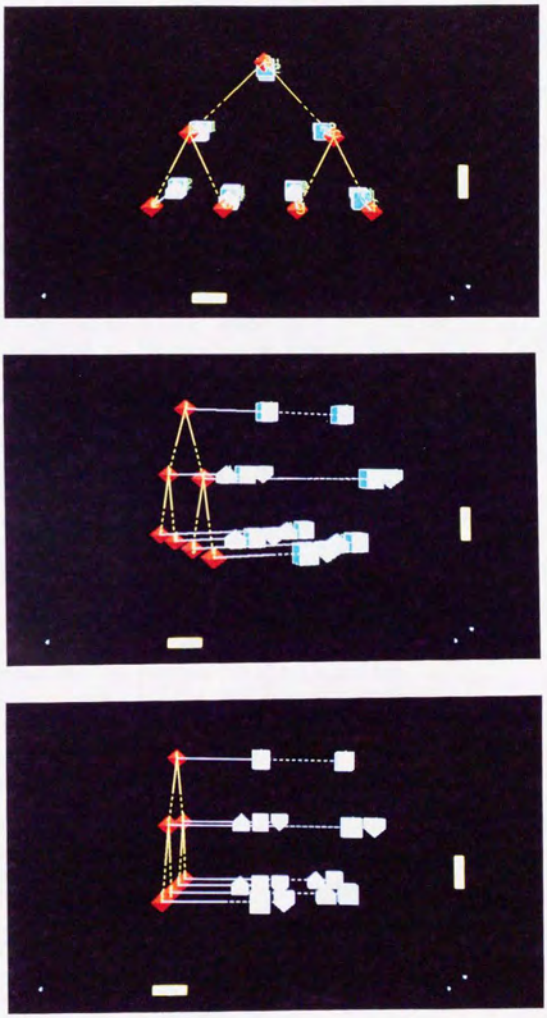


Figure 7.32: Flavor に対応したブラウザ

```
(defclass a () ())
(defclass b (a) ())
(defclass c (a) ())
(defclass d (b) ())
(defclass e (b) ())
(defclass f (c) ())
(defclass g (c) ())

(defmethod m1 ((a a)) (print 8))
(defmethod m1 ((b b)) (print 5))
(defmethod m1 :after ((b b)) (print 4))
(defmethod m1 :before ((b b)) (print 6))
(defmethod m1 ((d d)) (print 3))
(defmethod m1 :after ((d d)) (print 7))
(defmethod m1 :before ((d d)) (print 1))
(defmethod m1 ((f f)) (print 2))
(defmethod m1 :after ((f f)) (print 2))

(defmethod m2 ((a a)) (print 6))
(defmethod m2 ((c c)) (print 4))
(defmethod m2 :after ((c c)) (print 9))
(defmethod m2 ((g g)) (print 3))
(defmethod m2 :before ((g g)) (print 7))
(defmethod m2 ((e e)) (print 2))
(defmethod m2 :before ((e e)) (print 0))
```

Figure 7.33: 実験に用いたクラス

```
(m1 (make-instance 'f))
(m2 (make-instance 'g))
(m1 (make-instance 'd))
(m2 (make-instance 'e))
```

Figure 7.34: 実験に用いたタスク

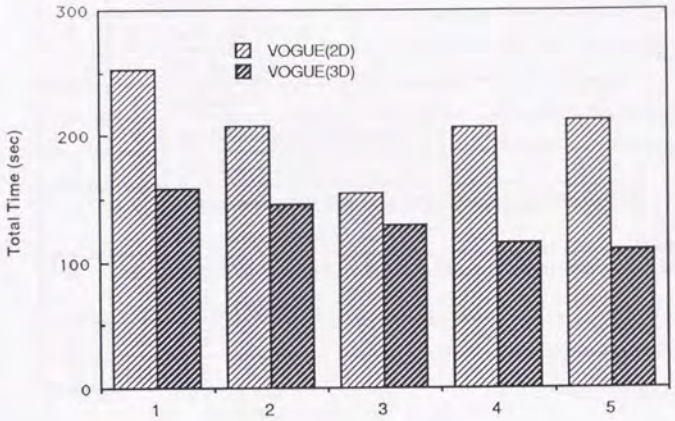


Figure 7.35: 個人別実験結果

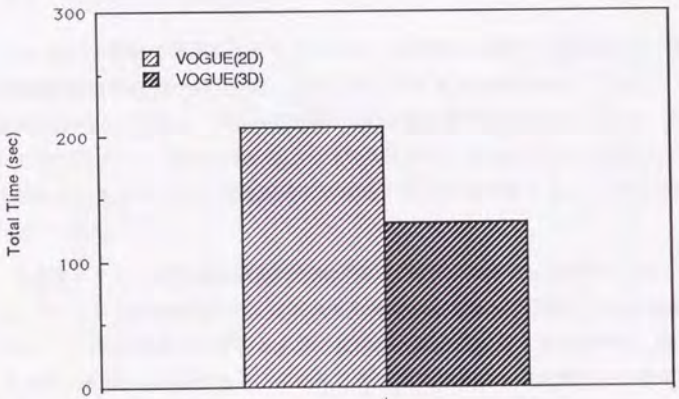


Figure 7.36: 実験結果の平均

重複度	個数	メソッド
10	1	size:
9	3	add:, do:, at:put:,
8	2	at:, storeOn:
7	1	collect:
6	2	printOn:, remove:ifAbsent:
5	2	\ =, copyFrom:to:
4	7	last:, select:, copyReplaceFrom:to:with:, occurencesOf:, first:, copy:, grow:
3	8	includes:, addLast:, addAll:, asString:, swap:with:, addFirst:, hash:, reverseDo:,
2	36	省略
1	123	省略

Table 7.1: メソッドの重複度

7.6 大規模データへの対応

7.6.1 大規模データでの問題点

ここまでの実験で使用したクラス・メソッドの例は比較的小規模なものであり、情報削減手法は不必要であったが、現実のオブジェクト指向言語のクラス・メソッドの数ははるかに大規模で、何らかの制御手法が必要な事は容易に予想できる。例えば Smalltalk-80 におけるクラス Collection 以下の代表的なサブクラスは Figure 7.37に示すように 15 個あり、そのメソッド数は総数 185 種類、計 329 個存在する。 これらの重複度を Table 7.1に示す。

この全メソッドを情報削減を行なわずに VOGUE で表示したのが Figure 7.38である。メソッドの数が多過ぎて、メソッドを特定することが不可能である。また、メソッド・ノード間の間隔を広げるとグラフ全体が z 軸方向に長くなり過ぎて、視点の変更 (スクロール等) が困難である。さらに、表示オブジェクト数の多さのためグラフィックの反応が極度に悪くなり、視点の変更その他の操作は無理である。結果として何らかの図形削減手法が必要である。

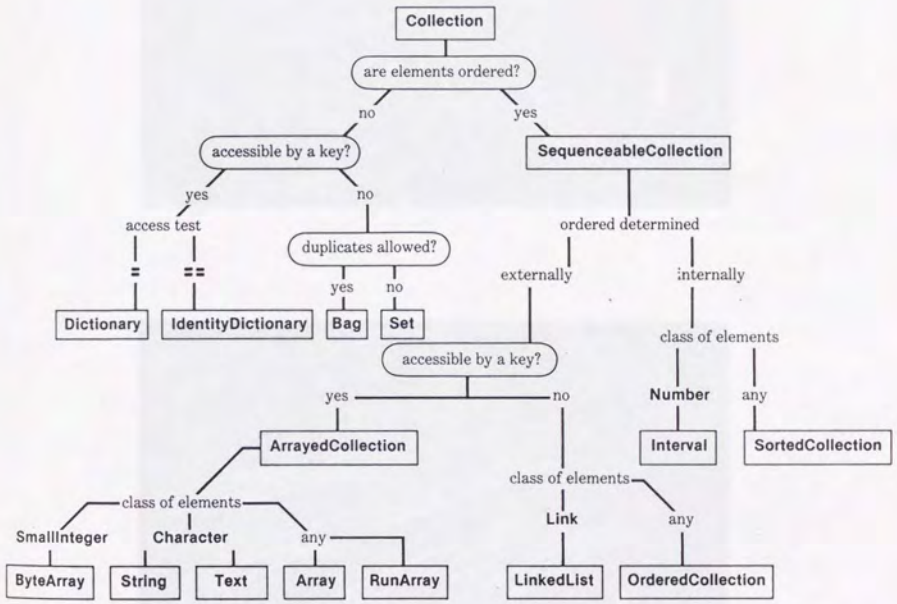


Figure 7.37: Smalltalk-80 の Collection のサブクラス

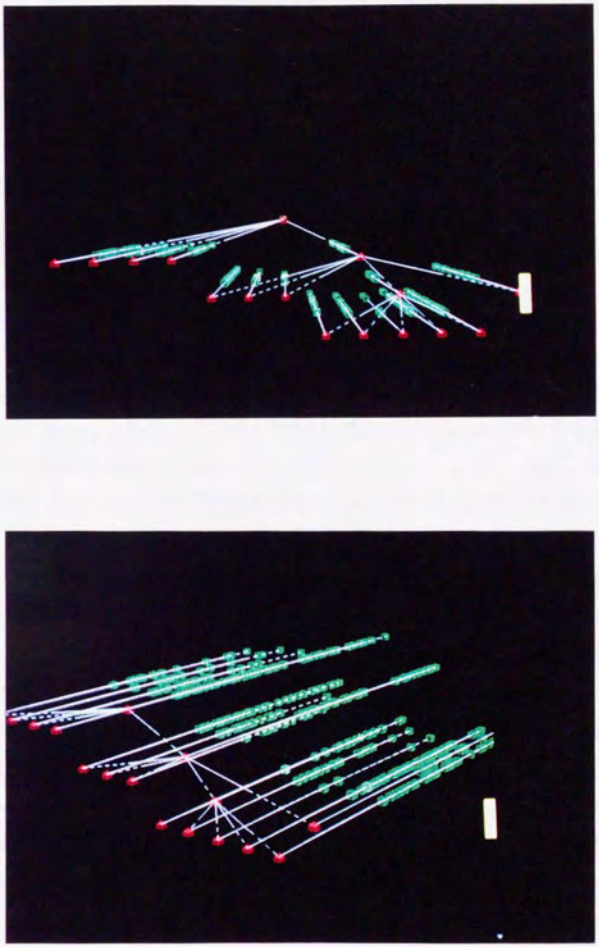


Figure 7.38: VOGUE による Smalltalk-80 の視覚化

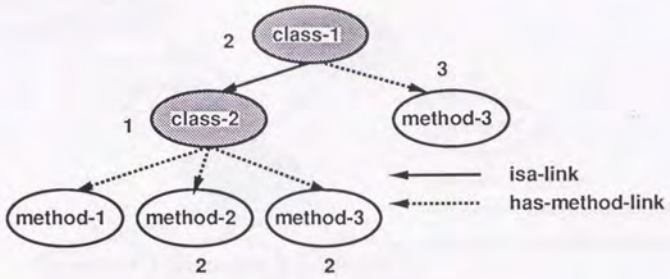


Figure 7.39: method-1 からの論理的距離

あるクラスに着目する場合、そのクラスに属する各メソッドとの論理的距離は等しく 1 である。従って、木の深さに基づく図形制御法でも Fractal View でもそれらのメソッドは全部表示されるかされないかのどちらかである。よって、このような場合には VOGUE における図形削減手法は役にたたない。クラスに着目しメソッドを探索する場合には、従来型のシステムブラウザで十分である。

次に、任意のメソッドに着目する場合、このメソッドと同じクラスに属するメソッドとの論理的距離は 2 であり、その他のメソッドとの距離は 3 以上である (Figure 7.39)。従ってこの場合にも、木の深さに基づく図形制御法でも Fractal View でもそれらのメソッドは全部表示されるかされないかのどちらかである。

この問題を解決するには、データベースに別な関係を定義することが必要である。特に、繰り返し述べてきている同じ名前のメソッドに着目した視点をより強調するような関係が必要となる。よって、ここでは CLOS で採用された総称関数の概念を、一般のオブジェクト指向言語におけるメソッド同士の関係にも採用することで、この問題の解決をはかる。

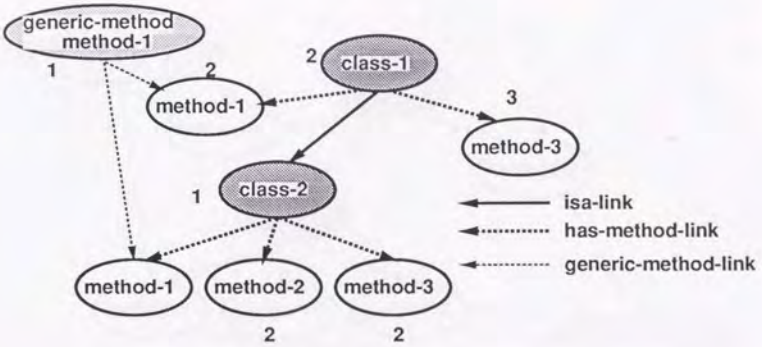


Figure 7.40: method-1 からの論理的距離

7.6.2 総称関数の概念の導入

CLOS で採用された総称関数の概念とは、従来のオブジェクト指向言語でのメソッドは対応するクラスに属するとした考え方に対し、同一名称をもったメソッドは総称関数によって束ねられるという考え方である。この概念を取り入れることにより、同一名称を持ったメソッド間の関係を強く表現することが可能となり、メソッドの表示をより効果的に表現することができる。つまり、同一名称を持ったメソッドを代表するようなノードを 1 個作り、これらのメソッドと接続することで、任意のメソッドに対しそれと同一名称のメソッドとの論理的距離を全て均等に 2 とすることができる (Figure 7.40)。結果として任意のメソッドから 2 の距離にあるノードは、同一名称をもったメソッド全部と同一クラスに属するメソッド全部ということになるが、一般にあるクラスに属する全メソッド数より同一名称のメソッド数の方が少ないので、任意のメソッドをルートとする論理木を考えた時、Fractal View の手法に基づいた値の伝播では同一名称のメソッドの集合に対して与えられる値の方が大きくなり、閾値を操作することでクラスに属するメソッド集合全部を表示せずに同一名称メソッドの集合を表示することが可能となる。

この関係の導入だけでもある程度の目的は達成されるが、ここではより実用的な視覚表現を得るためにもう一つの関係を定義する。あるメソッド内から他のメソッド呼出しが行なわれる場合、オブジェクト指向言語では起動されるメソッドを静的に特定

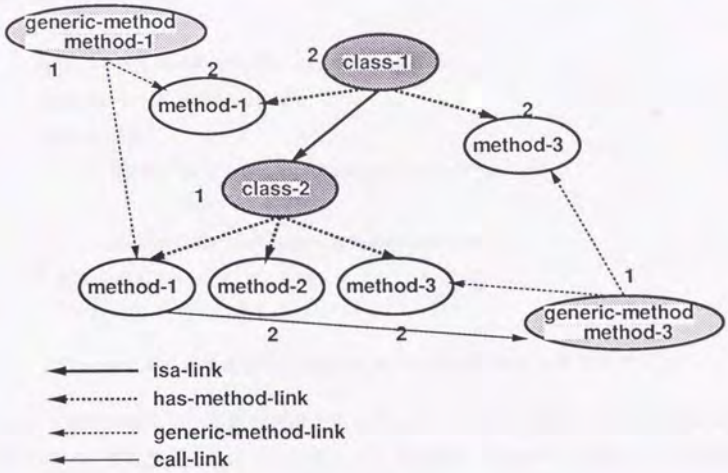


Figure 7.41: method-1 からの論理的距離

することは不可能であるが、これを総称関数への呼び出しと解釈すれば[2]、メソッド間の呼び出し関係をリンクとして定義できる。あるメソッドが呼び出しているメソッドは、他のメソッドに比べ重要性が高いはずであり、こうした関係の記述により、呼び出し関係のあるメソッドにはより高い重要度を与えることが可能となる。Figure 7.41は class-2 の method-1 が class-1 の method-3 を呼び出していると考えた場合のインスタンス間の関係である。

例えば Figure 7.42は Smalltalk-80 におけるクラス Dictionary のメソッド add: のプログラムであるが、このメソッドは例えば basicAt: や findKeyOrNil: を呼び出している。これを総称関数 basicAt: と findKeyOrNil: への呼び出しと捉えることにする。

以上の考察に基づきモデルに以下のような修正を加える。まず総称関数を generic-method クラスとして定義し、総称関数と同一名称を持つメソッドはこの generic-method クラスのインスタンスと generic-method-link で結合する。さらにあるメソッドとその

[2]実際 CLOS ではこのように解釈している

```
add: anAssociation
    | index element |
    index ← self findKeyOrNil: anAssociation key.
    element ← self basicAt: index.
    element isNil
            ifTrue: [self basicAt: index put: anAssociation.
                    tally ← + 1]
            ifFalse: [element value: anAssociation value].
    ↑ anAssociation
```

Figure 7.42: クラス Dictionary のメソッド add: のソースプログラム

メソッドが呼び出している総称関数とを call-method-link で結合する。具体的に新しくデータベースに登録したクラス定義とリンク定義とデータベース内のクラス階層を Figure 7.44に示す。

こうして実現されたのが Figure 7.45である。図は先に述べたクラス Dictionary のメソッド add: を選択している状態である。ただし call-method-link は、この add: から総称関数 findKeyOrNil: と総称関数 basicAt: への 2 本だけである。フラクタル次元を 2.5 とし、閾値を操作することで注目するメソッドに対し、同一名称を持つメソッド、及び呼び出し関係にあるメソッドが表示される。

```
(define-class generic-method () ())
(define-link generic-method-link ()())
(define-link call-method-link ()())
```

Figure 7.43: 総称関数に対応したクラス定義

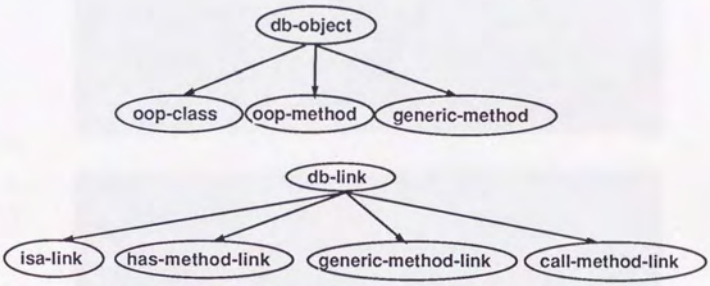


Figure 7.44: 総称関数に対応したクラス階層

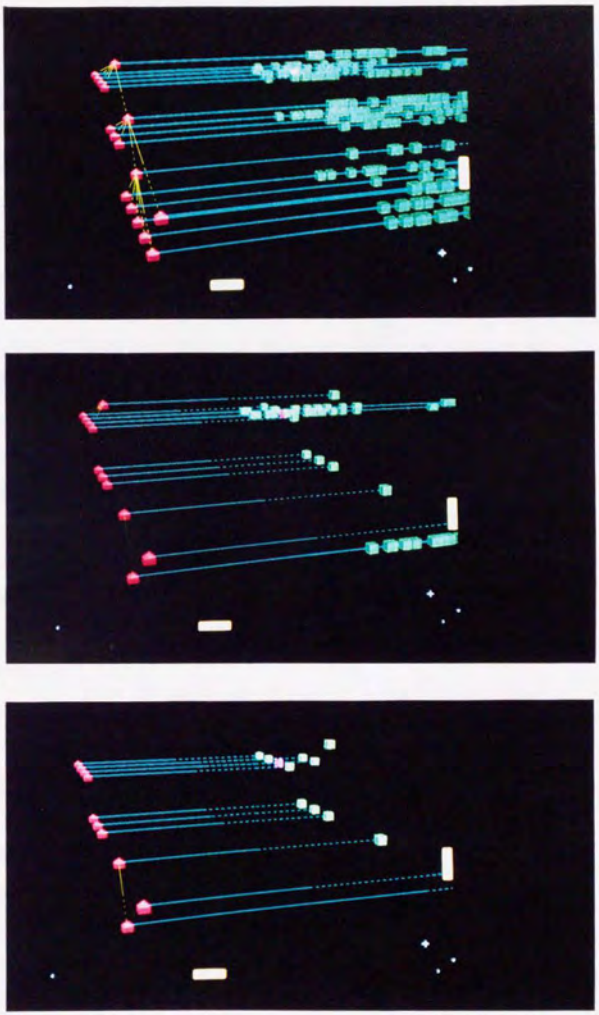


Figure 7.45: Fractal View による図形削減

7.7 本章のまとめ

本章では、オブジェクト指向言語のクラス・ライブラリの視覚化を取り上げ、VOGUEの適用を試みた。オブジェクト指向プログラミングにおけるクラス階層とメソッド継承という、互いに関連のある2つの異なる視点を持っており、3次元視覚化の例題として適切なものであると考える。

VOGUEでは、クラス及びメソッドをそれぞれノードとして描画し、クラス階層をxy平面に木構造として配置し、メソッドには所属するクラスと等しいx、y座標を与え、かつ同一名を持つメソッド同士は等しいz座標を持つようにすることで2つの視点をともに満足する視覚化表現を得た。さらに、得られた3次元視覚化表現の有効性を示すために、実際にSmalltalk-80上に実現されているクラス階層ブラウザを参考に、その本質を変化させないよう留意しつつ比較システムを作成し、3次元ツールとの比較実験を被験者を用いて行なった。その結果、オブジェクト指向言語のデバッグ時に直面するメソッド探索という課題に対して、明らかに3次元ツールの方がそのメソッド検索速度が速いことが示された。メソッド探索においては、クラス階層を意識しつつメソッドを探索する作業が必要となる。3次元表現を利用すれば従来のクラス階層図に対し、同一名のメソッドをz軸方向に整列させることが可能で、その結果としてユーザはメソッド探索における上への視点の移動と階層構造を同時に図としてみることができるため、その記憶の負荷、及びアクセス・スピードの点で従来ツールを上回っている。

さらに、より複雑な例としてデーモン・メソッドを持つFlavorを取り上げ、同様に視覚化を行なった。デーモン・メソッドを持たないオブジェクト指向言語の場合、メソッド探索は実際に起動される1つのメソッドを同定することで達成されるが、beforeメソッド、afterメソッドを持つFlavorの場合、着目するインスタンスの属するクラスの上位クラス全てのデーモン・メソッドが起動されるため、そのトレース作業は繁雑をきわめ、従来の2次元ツールでは対応不可能であると考えられる。一方、3次元視覚表現を用いると各デーモンは標準メソッドに対し、わずかにずらした位置に表示することで対応可能である。得られた3次元視覚化表現により、デーモンを含む複雑なメソッド継承が視覚的パターンとして、容易に把握可能であることが示された。

図形ツールにはこうした長所がある半面、非着目点の消去、つまり抽象化の機能が

不足している。実際の大規模なクラス・ライブラリへの対応を考えた場合これは無視できない問題である。例としてあげた Smalltalk-80 のクラス・ライブラリは、そのままでは表示することはできても、メソッドを特定することさえできない。VOGUE ではこれを CLOS で導入された総称関数の概念と第 4 章で開発した Fractal View を用いることで回避した。閾値を適当に設定することで、着目するメソッドとその近傍が表示された。

ただし、このように情報を整理し必要な所だけを見る目的には、Smalltalk-80 のシステム・ブラウザに代表されるメニュー形式のツールが優れている場合が多い。また、オブジェクト指向言語における各オブジェクト間のようにデータ間の関係がはっきりしている場合には、指定した部分だけを表示する方が効果的である。つまり、現実的な対応としては、この両者を用いシステム・ブラウザで素早くアクセスし、3 次元ツールでメンタル・モデルを確認しながらデバッグを行なうのが適当だと思われる。

第 8 章

考察および展望

8.1 演繹オブジェクト指向データベースとの関連

VOGUE ではオブジェクト間の関係を表すのにリンクという概念を用いている。そしてグラフ上では、このリンクを線分として表示している。これは、グラファ・プロセス上にリンクに対応するメモリが確保されることを意味するが、前述したようにハードウェアの制約上、グラフ上に作成できるオブジェクトの数には限界がある。

記述される関係の中には、類似したものが多く冗長な部分が含まれており、例えば、第 7 章における総称関数を例にとると、同じ名前を持ったメソッドは全てある総称関数とリンクで結ばれている。各メソッドがそれぞれ 1 つの総称関数と結ばれているわけであるから、全体として 329 本の冗長なリンクを持っていることになる。この冗長さを解決するために以下のようなルールの記述が考えられる。

```
IF
   ?method has name "add:"
   ?generic-method has name "add:"
THEN
   ?method is member of ?generic-method
```

このルールを用いることによって、総称関数とメソッドとの関係を間接的に記述することができ、直接的にリンクを張る必要がなくなる。そして、Fractal View の計算の際には、モデルの関係を考慮するようにすればよいと考えられる。

この概念は現在、オブジェクト指向データベースへの推論機構の導入という形で、

演繹オブジェクト指向データベース (Deductive Object-Oriented Database) の研究が行なわれている。オブジェクトという形によるデータモデルの自然な記述と、Prolog に代表される論理型言語における推論機構の融合によって、より知的なデータベースの実現が期待される。

VOGUE へのこうした機能の追加は、データモデル記述能力をより強力にし、かつグラフィックスの負荷を小さくする効果があると考えられる。

8.2 人工現実感との融合 – VOGUE-AR –

現在、人工現実感 (Artificial Reality) と呼ばれる新たなユーザ・インタフェース技術が注目を集めている。計算機内に構築されたモデルをグラフィックス表示し、ユーザは特殊な入出力デバイスを利用することで、実際には存在しない対象物とのインタラクションを行なうことが可能である。

VOGUE-AR はこの人工現実感技術に基づくソフトウェア開発環境であり、本論文で実現した VOGUE に 3 次元インタフェース技術を組み合わせたものである。ここで言う 3 次元インタフェース技術とは具体的には以下のようなものである。

1. 3 次元対象の正確なポインティング
2. 3 次元空間における視点の自由な変更

本論文ではこの 3 次元インタフェース技術に関する議論を時期尚早と判断し扱わなかったが、本節では現在および将来的技術の発達を考慮に入れた上で、VOGUE-AR の概念を具体化することを試みる。

8.2.1 3 次元入力デバイス

3 次元入力デバイスとして現在考えられるのものには以下のようなものがある。

- 2 次元マウス
  現在利用されている 2 次元マウスを使用し、縦横への移動、およびボタンの ON/OFF 状態を 3 次元的動きに変換する。問題点としてはマウスの動きと空間上の移動との対応を取るのが困難なことである。
- フライング・マウス
  基本的に 2 次元マウスにポリマス・センサに代表される 3 次元センサを付けた構造であり、3 次元センサで位置を検出し、ボタンでアクションを起こす。
- DataGlove
  ポリマス・センサの付いたグローブ型入力デバイスで、ポリマスで空間上の位置と姿勢を検出し、5 指の動きを光ファイバケーブルによって検出できる。問題点は、センサの精度が悪いことである。

- Space Ball
  圧力センサを付けた球形のデバイスで、グラフィックスの並進・回転の各移動を球形のハンドルに対し行なうことで達成できる。

この他にも、力覚フィードバックの可能な入力デバイスを始め、多くの入力デバイスが研究開発中である。

ボリマス・センサは精度という点では問題があるが、仮想空間における姿勢と実際にユーザが指示する姿勢が同一であるという点で、インタフェースとしての自然さが評価できる。また、3 つのマウス・ボタンによって指示できるアクションは限られているが、手の形による指示はより多くの指示を行なうことができる。よって VOGUE-AR では Data Glove(Figure 8.1) をポインティング・デバイスとして採用する。

8.2.2 出力デバイス

図形ツールの問題点の 1 つとして、表示領域の問題があることは第 2 章で述べた。これに対する回答の 1 つはより大きなディスプレイを使用することである。現在、ディスプレイの大型化が進み、19 インチ程度のものが一般的に使用されている。また、液晶プロジェクタの価格も低下しつつあり、100 インチ程度の大きなディスプレイの利用も程なく開始される。これらに関する先駆的な研究としては、MIT の Media Lab. における Put That There があり、壁一面をディスプレイとして利用したインタフェースの研究が行なわれた。

これに対し現在頭部搭載型ディスプレイの研究が行なわれている。右目用と左目用の 2 個のディスプレイを搭載したゴーグルを頭部に装着し、それぞれ右目用および左目用画像が提示され立体視することができる。これにより視野が広がり表示領域も増加する。この頭部搭載型ディスプレイの重要な点は、ゴーグルに付けられたボリマス・センサが頭の位置・姿勢を検出し、この情報をモデルを保持する計算機にフィードバックすることで、左を向けば左方の絵、右を向けば右方の絵を見ることができる。よって視野はユーザの全周に広がり、かつこれまでスクロール・バーで行なっていた視点の移動をより自然な形で行なえることが重要である。欠点としては、ディスプレイの解像度が低いことと、グラフィックス以外が見えなくなることがあげられる。

VOGUE-AR は図形情報とともにテキスト情報をも扱わなければならない。よって出力デバイスとしては透過型頭部搭載ディスプレイ (STHMD)(Figure 8.2) を採用す

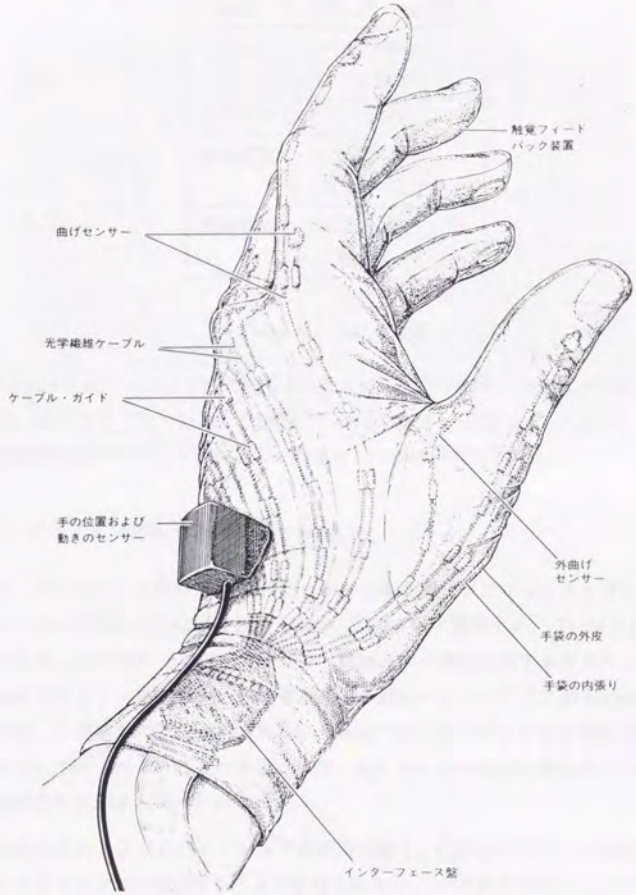


Figure 8.1: データグローブ

名称	IRIS-4D/210VGX
CPU	R3000 R3010
クロック	25MH
演算速度	20MIPS 3.3MFLOPS
描画性能	100 万ベクタ / 秒 100 万ポリゴン / 秒

Table 8.1: Iris の仕様

る。STHMD はハーフミラーを利用することで、仮想空間と現実空間の融合を行なえる [48]。従来のディスプレイとの併用で、視覚的表示は仮想空間上に表示し、テキストの編集等は現実のディスプレイ上で行なうという操作が可能となる。

8.2.3 グラフィックス・ワークステーション

また、データベース部分は VOGUE と同一であるが、グラフィック・ワークステーションとして Silicon Graphics 社の Iris4D/210VGX を採用する。Table 8.1が Iris の仕様である。HP9000 とのグラフィックス描画速度の差を比較するために、描画されているオブジェクト数を横軸、画面更新速度を縦軸にとったグラフが Figure 8.4である。また、メモリ等の制約から HP9000 で描画できるオブジェクトの数は 10^3 程度であったが、Iris では 10^4 程度になる。実際、第 6 章における電力制御用トレースデータの視覚化には Iris が用いられている。

この例にもあるようにハードウェアの進歩は速く、数年後にはさらに高機能なグラフィックス・マシンが登場することが予想されるが、VOGUE ではハードウェア依存の基本的なグラフィックス部分のみを Grapher 本体で実現し、モデル管理、拡張機能を分離しているため、こうした技術の進歩にも即座に対応できる。

以上をまとめた VOGUE-AR のシステム構成図を Figure 8.5に示す。VOGUE-AR の実現によって、以下のメリットがあると考えられる。

- グラフィックス表示領域の拡大

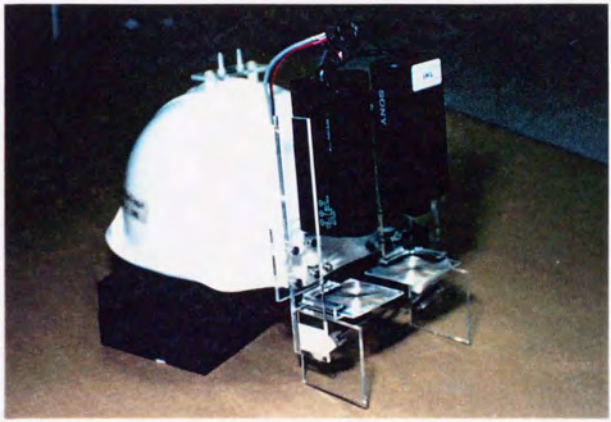


Figure 8.2: 透過型頭部搭載ディスプレイ

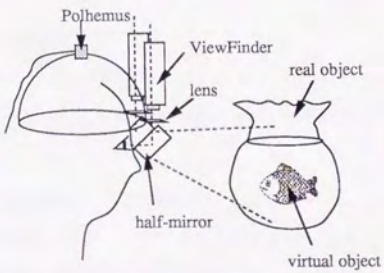


Figure 8.3: 透過型頭部搭載ディスプレイの概念図

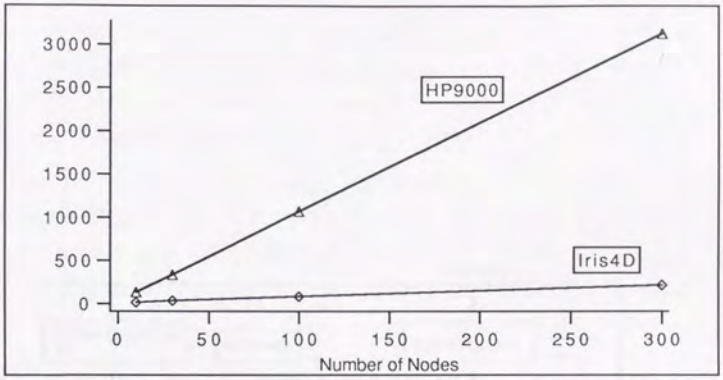


Figure 8.4: HP と Iris の画面更新速度の違い

- 手を用いた自然なインタラクション
- 深部知覚[1]による記憶の効率化

実用化に向けて今後研究されなければならない点としては以下の点があげられる。

- 入出力デバイスの精度向上
  現在のポリマス・センサや光ファイバ・ケーブルでは精度の点で問題が多い。また、出力デバイスとしてはより軽量で装着を意識させないものが必要であろう。
- 3 次元環境での基本アクションの整理
  2 次元環境におけるマウス・クリック、メニュー・プルダウン、アイコンのドラッグに相当する概念を整理する必要がある。

[1]対象物の位置を腕の伸ばし具合等の体全体で記憶すること

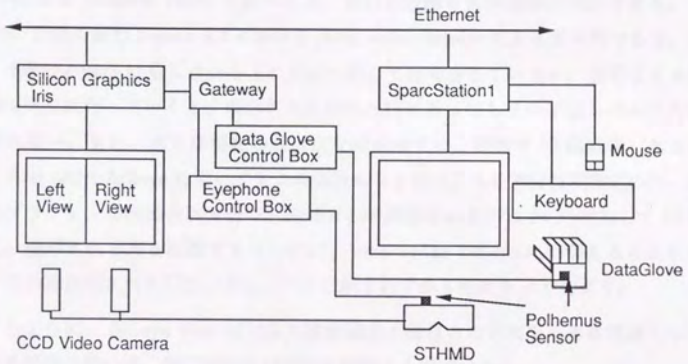


Figure 8.5: VOGUE-AR のシステム構成図

8.3 Generalized Fractal Views の応用

第 4 章で開発した Generalized Fractal Views は、本論文においては図形の制御に用いられたが、Generalized Fractal Views は着目点とそこをルートとする木が定義できるなら他の情報構造に対しても適用可能である。ここでは他への応用例として、文献 [36] において Generalized Fisheye Views の適用例として取り上げられた C プログラムにおける表示量制御を考えてみる (Figure 8.6)。

54 行のプログラムに対して、例えば 23 行表示可能なエディタを利用すると一般の表示方法では着目行を中心として上下各々 11 行ずつが表示されることになる。一方、Generalized Fisheye Views を用いると、着目点近傍と大局情報が表示できる。Figure 8.7は第 4 章の Figure 4.2 における first order fisheye による表示例である。しかし、fisheye view は着目点から下の分岐に関しては考慮していない。着目点を 6 行目の while 文に持っていくと、表示される行は 8 行になってしまい下位レベルの行は表示されない。また、例えば現在の着目点を変更せずに、関数を 10 個定義したとすると、first order fisheye によって 1 つの関数につき表示される部分は関数名の行、左右の各ブラケットの行の合計 3 行で、全体として表示される行は 30 行増加して 53 行となる。表示される行を削減するためには、zero order fisheye に切替えることになるが、この時表示される行は、Figure 8.8に示すわずか 6 行になってしまう。

このように、fisheye view は対象の階層構造と着目点の近傍をともに考慮しつつ表示する機能は持つが、表示情報を自動的に制御する機能はない。

```
#define DIG 40
#include <stdio.h>
main()
{
    int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
    while((c=getchar()) != EOF){
        if(c >= '0' && c<= '9'){
            x[0] = 10 * x[0] + (c-'0');
            for (i=1;i<k;i++){
                x[i]=10*x[i]+x[i-1]/10000;
                x[i-1] %= 10000;
            }
        }else{
            switch(c){
                case '+':
                    t[0] = t[0] + x[0];
                    for(i=1;i<k;i++){
                        t[i]= t[i] + x[i] + t[i-1]/10000;
                        t[i-1] %= 10000;
                    }
                    t[k-1] %= 10000;
                    break;
                case '-':
                    t[0] = (t[0] + 10000) - x[0];
                    for(i=1;i<k;i++){
                        t[i] = (t[i] + 10000) - x[i] - (1 - t[i-1]/10000);
                        t[i-1] %=10000;
                    }
                    t[k-1] %= 10000;
                    break;
                case  'e':
                    for(i =0;i<k;i++) t[i]= x[i];
                    break
                case 'q':
                    exit(0);
                default:
                    noprint = 1;
                    break;
            }
            if(!noprint){
                for(i=k-1;t[i] <= 0 && i> 0; i--);
                printf("%d",t[i]);
                if(i > 0) {
                    for (i --; i>= 0; i--){
                        printf("%04d",t[i]);
                    }
                }
                putchar('\n');
                for (i=0; i> k; i++) x[i] = 0;
            }
        }
        noprint = 0;
    }
}
```

Figure 8.6: 対象とする C プログラム

```
 1    #define DIG 40
 2    #include <stdio.h>
 3    main()
 4    {
 5     int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
 6     while((c=getchar()) != EOF){
 7         if(c >= '0' && c<= '9'){
13          }else{
14              switch(c){
15                  case '+':
23                  case '-':
31                  case  'e':
32                      for(i =0;i<k;i++) t[i]= x[i];
33                      break
34                  case 'q':
36                  default:
39              }
40              if(!noprint){
50              }
51          }
52          noprint = 0;
53      }
54   }
```

Figure 8.7: 32 行目からの first order fisheye

```
 4    {
 6     while((c=getchar()) != EOF){
13          }else{
14              switch(c){
31                  case  'e':
32                      for(i =0;i<k;i++) t[i]= x[i];
```

Figure 8.8: 32 行目からの zero order fisheye

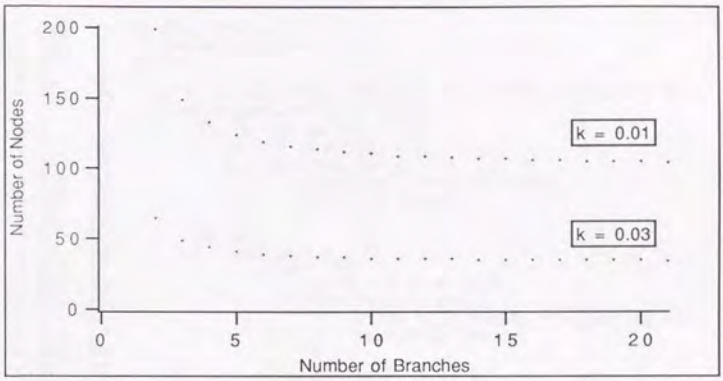


Figure 8.9: 分岐数と表示されるノードの関係

次に Fractal view で同様の削減を行なう。いまフラクタル次元を 1 に固定した時、第 4 章での式から得られる分岐数と表示情報量との関係を分岐数が 2 から 20 程度でグラフにプロットすると Figure 8.9のようになる。

より具体的に考察するために、Figure 8.6において 32 行目と 6 行目にそれぞれ着目した時、着目点を 1 として

$$r = N^{-\frac{1}{D}}$$

に基づき各ノードに伝播される値を Figure 8.10に示す。各行の先頭が 32 行目に着目した時の伝播値、次が 6 行目に着目した時の伝播値、3 番目が行番号である。この値に基づき、ある閾値を設定した場合表示されるノード数は Table 8.2、Table 8.3に示すとおりである。

閾値を変化させることによって表示される行がどのように変化するかを、各々 Figure 8.11から Figure 8.18に示す。

閾値の変化に対して、表示される行数が徐々に変化させることができる。さらにこれらの閾値と表示される行数の関係を Figure 8.19に示す。このグラフから、ある閾値に対して表示される行数は 6 行目に着目するときも 32 行目に着目するときもあまり

```
0.0005 0.017   1  #define DIG 40
0.0005 0.017   2  #include <stdio.h>
0.0005 0.017   3  main()
0.006  0.2     4  {
0.002  0.07    5   int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
0.025  1       6   while((c=getchar()) != EOF){
0.006  0.2     7       if(c >= '0' && c<= '9'){
0.002  0.07    8           x[0] = 10 * x[0] + (c-'0');
0.002  0.07    9           for (i=1;i<k;i++){
0.001  0.035  10                 x[i]=10*x[i]+x[i-1]/10000;
0.001  0.035  11                 x[i-1] %= 10000;
0.002  0.07   12            }
0.1    0.2    13        }else{
0.5    0.05   14           switch(c){
0.1    0.0 1  15               case '+':
0.02   0.002  16                   t[0] = t[0] + x[0];
0.02   0.002  17                   for(i=1;i<k;i++){
0.01   0.001  18                       t[i]= t[i] + x[i] + t[i-1]/10000;
0.01   0.001  19                       t[i-1] %= 10000;
0.02   0.002  20                   }
0.02   0.002  21                   t[k-1] %= 10000;
0.02   0.002  22                   break;
0.1    0.01   23               case '-':
0.02   0.002  24                   t[0] = (t[0] + 10000) - x[0];
0.02   0.002  25                   for(i=1;i<k;i++){
0.01   0.001  26                      t[i]=(t[i]+10000)-x[i]-(1-t[i-1]/10000);
0.01   0.001  27                      t[i-1] %=10000;
0.02   0.002  28                   }
0.02   0.002  29                   t[k-1] %= 10000;
0.02   0.002  30                   break;
1      0.01   31               case 'e':
1      0.005  32                   for(i =0;i<k;i++) t[i]= x[i];
0.5    0.005  33                   break
0.1    0.01   34               case 'q':
0.1    0.01   35                   exit(0);
0.1    0.01   36               default:
0.05   0.005  37                   noprint = 1;
0.05   0.005  38                   break;
0.025  0.05   39           }
0.025  0.05   40           if(!noprint){
0.004  0.008  41               for(i=k-1;t[i] <= 0 && i> 0; i--);
0.004  0.008  42               printf("%d",t[i]);
0.004  0.008  43               if(i > 0) {
0.002  0.004  44                   for (i --; i>= 0; i--){
0.002  0.004  45                       printf("%04d",t[i]);
0.002  0.004  46                   }
0.004  0.008  47               }
0.004  0.008  48               putchar('\n');
0.004  0.008  49               for (i=0; i> k; i++) x[i] = 0;
0.025  0.05   50           }
0.006  0.2    51        }
0.006  0.2    52        noprint = 0;
0.002  0.07   53    }
0.0005 0.017  54  }
```

Figure 8.10: 各行に伝播される値

閾値	表示される行数
1	2
0.5	4
0.1	10
0.05	12
0.025	16
0.02	26
0.01	30

Table 8.2: 32 行目に着目した時の閾値と 表示される行数の関係

閾値	表示される行数
1	1
0.2	6
0.07	11
0.05	15
0.035	17
0.017	22
0.01	28

Table 8.3: 6 行目に着目した時の閾値と 表示される行数の関係

```
 6        while((c=getchar()) != EOF){
13            }else{
14                switch(c){
15                    case '+':
16                        t[0] = t[0] + x[0];
17                        for(i=1;i<k;i++){
18                            t[i]= t[i] + x[i] + t[i-1]/10000;
19                            t[i-1] %= 10000;
20                        }
21                        t[k-1] %= 10000;
22                        break;
23                    case '-':
24                        t[0] = (t[0] + 10000) - x[0];
25                        for(i=1;i<k;i++){
26                            t[i] = (t[i] + 10000) - x[i] - (1 - t[i-1]/10000);
27                            t[i-1] %=10000;
28                        }
29                        t[k-1] %= 10000;
30                        break;
31                    case  'e':
>>32                     for(i =0;i<k;i++) t[i]= x[i];
33                      break
34                    case 'q':
35                        exit(0);
36                    default:
37                        noprint = 1;
38                        break;
39                }
40                if(!noprint){
50                }
```

Figure 8.11: 32 行目に着目 (閾値 =0.01)

```
 6        while((c=getchar()) != EOF){
13            }else{
14                switch(c){
15                    case '+':
16                        t[0] = t[0] + x[0];
17                        for(i=1;i<k;i++){
20                        }
21                        t[k-1] %= 10000;
22                        break;
23                    case '-':
24                        t[0] = (t[0] + 10000) - x[0];
25                        for(i=1;i<k;i++){
28                        }
29                        t[k-1] %= 10000;
30                        break;
31                    case  'e':
>>32                     for(i =0;i<k;i++) t[i]= x[i];
33                      break
34                    case 'q':
35                        exit(0);
36                    default:
37                        noprint = 1;
38                        break;
39                }
40                if(!noprint){
50                }
```

Figure 8.12: 32 行目に着目 (閾値 =0.02)

```
 6        while((c=getchar()) != EOF){
13            }else{
14                switch(c){
15                    case '+':
23                    case '-':
31                    case 'e':
>>32                      for(i =0;i<k;i++) t[i]= x[i];
33                        break
34                    case 'q':
35                        exit(0);
36                    default:
37                        noprint = 1;
38                        break;
39                }
40                if(!noprint){
50                }
```

Figure 8.13: 32 行目に着目 (閾値 =0.025)

```
13            }else{
14                switch(c){
15                    case '+':
23                    case '-':
31                    case 'e':
>>32                      for(i =0;i<k;i++) t[i]= x[i];
33                        break
34                    case 'q':
35                        exit(0);
36                    default:
37                        noprint = 1;
38                        break;
```

Figure 8.14: 32 行目に着目 (閾値 =0.05)

```
    1    #define DIG 40
    2    #include <stdio.h>
    3    main()
    4    {
    5        int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
>>  6        while((c=getchar()) != EOF){
    7            if(c >= '0' && c<= '9'){
    8                x[0] = 10 * x[0] + (c-'0');
    9                for (i=1;i<k;i++){
   10                    x[i]=10*x[i]+x[i-1]/10000;
   11                    x[i-1] %= 10000;
   12                }
   13            }else{
   14                switch(c){
   15                    case '+':
   23                    case '-':
   31                    case  'e':
   34                    case 'q':
   35                        exit(0);
   36                    default:
   39                }
   40                if(!noprint){
   50                }
   51            }
   52            noprint = 0;
   53        }
   54    }
```

Figure 8.15: 6 行目に着目 (閾値 =0.01)

```
    1    #define DIG 40
    2    #include <stdio.h>
    3    main()
    4    {
    5        int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
>>  6        while((c=getchar()) != EOF){
    7            if(c >= '0' && c<= '9'){
    8                x[0] = 10 * x[0] + (c-'0');
    9                for (i=1;i<k;i++){
   10                    x[i]=10*x[i]+x[i-1]/10000;
   11                    x[i-1] %= 10000;
   12                }
   13            }else{
   14                switch(c){
   39                }
   40                if(!noprint){
   50                }
   51            }
   52            noprint = 0;
   53        }
   54    }
```

Figure 8.16: 6 行目に着目 (閾値 =0.017)

```
   4    {
   5        int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
>> 6        while((c=getchar()) != EOF){
   7            if(c >= '0' && c<= '9'){
   8                x[0] = 10 * x[0] + (c-'0');
   9                for (i=1;i<k;i++){
  10                    x[i]=10*x[i]+x[i-1]/10000;
  11                    x[i-1] %= 10000;
  12                }
  13            }else{
  14                switch(c){
  39                }
  40                if(!noprint){
  50                }
  51            }
  52            noprint = 0;
  53        }
```

Figure 8.17: 6 行目に着目 (閾値 =0.035)

```
   4    {
   5        int c, i, x[DIG/4], t[DIG/4], k = DIG/4, noprint = 0;
>> 6        while((c=getchar()) != EOF){
   7            if(c >= '0' && c<= '9'){
   8                x[0] = 10 * x[0] + (c-'0');
   9                for (i=1;i<k;i++){
  12                }
  13            }else{
  14                switch(c){
  39                }
  40                if(!noprint){
  50                }
  51            }
  52            noprint = 0;
  53        }
```

Figure 8.18: 6 行目に着目 (閾値 =0.05)

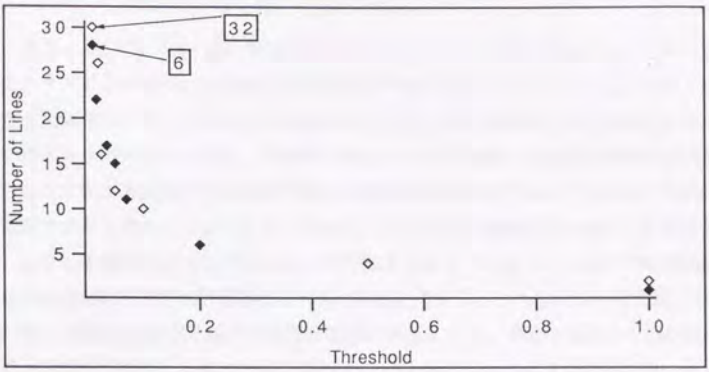


Figure 8.19: 閾値と表示されるノード数の関係

差がないことがわかる。例えば閾値として 0.01 を採用すると表示行数は約 30 行程度となる。

また、10 個の関数定義が存在する場合には、先に述べたように関数 1 個につき 3 行増えるため、6 行目、32 行目のどちらに着目点がある場合でも、1、2、3、54 行の 4 行に伝播されている値が小さくなる。上で示した表示例には影響がなく、また例えば 3 行目に着目しているとしても表示される行数はある範囲内に制御される。

このように Fractal View は Fisheye View と異なり、閾値の変化による表示情報量の柔軟な設定が可能であり、かつ着目点変更による提示情報量の変化を極力おさえることが可能な汎用的情報制御手法である。従ってその適用範囲は広く、 例えば、アウトラインプロセッサ (HyperText) 等にも適用可能であると思われる。また、本論文で何度も述べているように、いかに高機能なグラフィックス・ワークステーションを利用しようとも表示図形数と画面更新スピードは比例し、従って表示できる図形数には必ず上限が存在する。人工現実感システム等においても、仮想環境の記述をよりリアルにしようとするほど表示図形数は増加し、システムの反応性、特に視点変更時の画面の追従性が悪くなることが予想される。一般に環境記述は第 5 章で述べたセグメントを利用して階層的に記述されるが、本論文で開発した Fractal View をファーム

ウェア化して、グラフィックス・マシンに搭載し、着目するセグメントからの Fractal View をサポートすれば、閾値の設定で表示セグメント数を一定に保つことが可能で、システムの反応性も一定となると考えられる。

この Fractal View は 1 種の情報圧縮であると言える。実際、画像処理の分野においてはフラクタルに基づいた効果的な画像圧縮手法が提案されている [14]。ただしここで言う情報圧縮とは、UNIX の compress に代表される情報の正確な再現を目指すものではない。画像圧縮の場合、圧縮率を高くする程元画像の正確な再現率は低くなるが、たとえラフな画像でも受け取る側に必要な情報が存在すれば、それは十分に正しい圧縮であると言うことができる。よって、ここで言う情報圧縮の場合、その正しさは、必ずしも情報の正確な再現によって評価されるものではなく、受け取る側の要求を含めた相対的な尺度で評価されるべきであると考える。この意味から言えば、Fractal View は閾値の変更で提示情報量の制御ができるため、情報の受け取る側の要求に応じた圧縮を行なうことができる。よって、Fractal View による情報圧縮は、正しい圧縮であるとの評価ができると思われる。

8.4 情報、制約条件、形

生物は遺伝子の DNA に全ての情報が記録されている。この DNA の持つ情報に対して、ある制約条件と確率的事象が加わって、生物の様々な形態をつくり出していく。これは形態発生 (Morphogenesis) と呼ばれている。DNA 情報と制約条件から作り出された形によって、我々は個体間の持つ情報の差異を認識できるわけである。

第3章でも述べたように、一般にソフトウェアが持つのは情報だけであり、表出特徴としての形状が存在しないため、我々は情報間の差異や情報の異常性に気づきにくく、これがソフトウェアに対する理解を妨げていると考えられる。従って、情報に対して適当な制約を与えることで形を作りだし、情報を視覚的物体として具現化させることができれば、この情報に対する理解度は促進されると考えられる。本論文では、例えばクラス階層は木構造に描画するという制約を与え、メソッドはクラスの xy 座標とメソッド名で整列した z 座標に配置するという制約を与え、3 次元形状を作り出した。ここでは、この情報と制約について考察を行なう。

例として、第6章で取り上げた分散マニピュレータの計算機に対応するノードの配置を考えてみる。第6章における視覚化では著者の主観に従い、比較的メッセージ送信のパターンが見やすいと思われる位置にこれらノードの配置を行なった。ただし、この配置を発見するのは試行錯誤的である。

ここで、VOGUE の機能の1つである焼きなまし手法の適用を考えてみる。いまノードの可動範囲を xy 平面だけに限定し、計算機同士で通信が行なわれている場合には、これらのノード間にリンクが存在するものとする。この状態で数回焼きなまし手法を試みると、常に同じ形状になるとは限らず幾つかの局所的安定点に収束するが、確率的に見るとある収束しやすい特定のパターンが存在する (Figure 8.20、Figure 8.21)。ここで特徴的なのは、このような簡単な制約によっても同じ情報からは同じ形状が形成されやすいという点と、この制約によって形成された形は、そのメッセージ送信の様子が非常に見やすいという点である。もちろん、この場合関節間の順序関係は考慮していないので、関節の順序を重視する立場から見ればこの配置は無意味であるが、この視点に対しては別な制約を考えれば解決できると思われる。

次に図を用いた類推という側面から考えることにする。Figure 8.22,Figure 8.23は文献 [37] から取った例で、各々太陽系に関する知識と原子に関する知識を表現している。いま、各々の知識が何らかの記述方式で知識ベースに蓄積されている時、これら

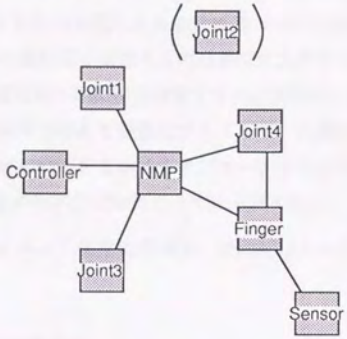


Figure 8.20: 焼きなましによって作られるパターン (1)

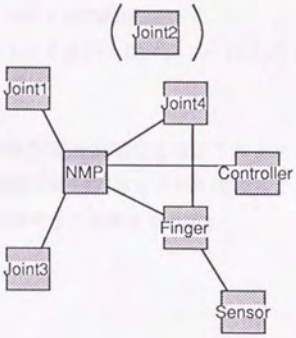


Figure 8.21: 焼きなましによって作られるパターン (2)

の知識からシステムが自動的に両者を照合し、両者の類似性を発見するシステムに関する研究が人工知能の分野において行なわれている。これは類推による学習と呼ばれる。こうしたシステムは、各知識ノードとノード間を接続する関係をシンボリックなパターン照合を行ないその類似性を判断し、複雑なメカニズムを必要とする。しかし、知能機械を実現する 1 つのアプローチとしてその研究に期待されるものも小さくない。ところで、例示した図はこれらの知識をある制約に基づいて視覚化したものと言えるが、人間がこの両者からその類似度を認識するのは非常にたやすい。逆に、このグラフが乱雑に描画されていれば両者の類似性を発見するのは困難だと思われる。このように、複雑な知識もある制約条件を与えて視覚化することで、人間の思考を支援することが可能である。前者が機械に全てをまかせるアプローチであるのに対して、本論文のアプローチは機械と人間との作業分担を行なっている例である。

本論文が示したソフトウェア視覚化手法は、制約記述という点で非力であった。これに対して例えば、

- 関係 (リンク) の強度の記述
  リンクに対して重みを付加することで、関係の記述能力を強化する。
- リンクの種類によるベクトルの記述
  ある関係を表現するリンクは例えばベクトル (1,1,0) の方向に描画するといった記述。

等が考えられる。より簡単に制約を記述できるようにシステムを発展させることで、情報に対する人間の接触形態や接触方法をさらに進めることができ、情報のさらなる利用が進められることが期待できると考えられる。

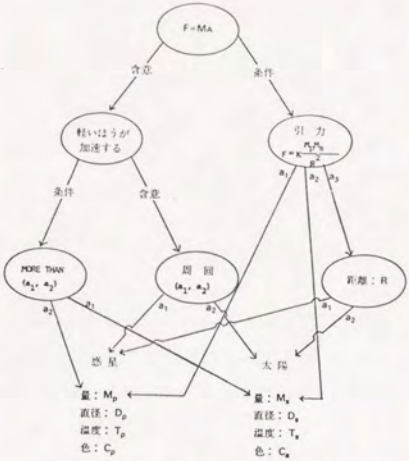


Figure 8.22: 太陽系に関する知識

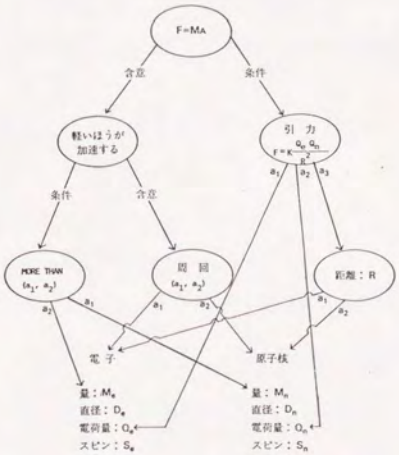


Figure 8.23: 原子に関する知識

第 9 章

結論

本論文では、ソフトウェア工学的立場から、3 次元 CG を用いた視覚化表示の利用法に関する考察を行ない、3 次元視覚化表示の提案と実用的視覚化システムを構築する上での問題点について議論を行なった。本研究で得られた結論を要約すると次のとおりである。

- ソフトウェア視覚化ツールに関する考察を行ない、実際のソフトウェアに適用する際の問題点を機能的および認知的側面から明らかにした。
- 3 次元 CG を用いた視覚化の提案を行ない、その利用法を以下のように整理した。
  - 弱い利用法
    グラフの視認性を向上させる
  - 強い利用法
    * 互いに干渉しあわない関係の分離
    * 2 次元的構造を持った対象のダイナミクスの表現

  さらに、ソフトウェア開発における視点のうち強い利用法の実例を示した。
- フラクタルの概念に基づく汎用的情報量制御手法を開発し、数値シミュレーションによってその有効性を示した。
- 提案した枠組に基づいたシステム VOGUE を試作し、実際のソフトウェア視覚化に適用した。

- 電力制御用ソフトウェアの視覚化においては、モジュール構造とタスク間通信の視覚化を行ない、図形数制御手法の実用性が示された。さらに、時間軸を導入した強い3次元化によりプロセスの実行状況を同時に視覚化でき、各プロセスの協調動作の視認性が向上した。
- オブジェクト指向言語のクラス・ライブラリの視覚化においては、クラス階層とメソッド継承の両視点を同時に視覚化できた。メソッド探索が視覚的に支援され、作業時間が短縮された。

どちらの例においても、3次元視覚化表現を利用することによって、従来の平面的な視覚化表現では支援することが困難な視点の提供が可能となった。また、図形数削減手法が有効であった。

以下、各章の結論をまとめる。

第1章

ソフトウェアの効率的開発のためには、ユーザのメンタル・モデル形成を支援することが重要であり、その手段としてソフトウェアを視覚化することが有効であると考えられる。ただし本研究では図形を従来のプログラミングを側面から支援するツールととらえる立場をとる。

第2章

図形言語に関するまとめを行ない、従来の視覚化ツールに関する考察を行なった。その結果、視覚化ツールが実際のソフトウェア開発に利用されるためには以下の点を考慮する必要がある。

- 表現手法としてはグラフ表現が汎用性の点で優れており、領域系図、座標系図への対応も可能である。
- 異なるソフトウェアに対応するためには、モデル化の機能としてデータベースが必要である。また、データモデルとしてはオブジェクト指向モデルが良い。
- 視覚化対象の量に対応するためには、高度なグラフィックス機能が必要である。
- 図形数の制御が必要である。図形数の削減は、人間にとって図の複雑さを削減するとともに、システムの反応性を高める。

- マルチウィンドウの乱用は有害である。

第3章
3次元CGを用いたソフトウェアの視覚化について考察を行なった。3次元CGの視覚化への導入は、大規模データへの対応、仮想表示、ダイナミクスの表現、コミュニケーションの円滑化をもたらすと考えられる。そして物理的形状を持たないソフトウェア情報での利用法を整理し、オブジェクト指向言語、並列プログラミング、版・構成管理、プロジェクト管理等の実例を挙げて示した。さらに、実システムを構築する上で考慮しなければならない点を述べた。それらは以下のようなものである。

- 視点の連続的移動の支援
- 3次元立体視の支援
- 図形数制御

第4章
汎用的情報量制御手法として、フラクタル次元に基づく制御手法を開発した。本手法の特徴は、全体としての表示量をほぼ一定に制御できる点とこの表示量をフレキシブルに設定できる点である。本手法の有効性は数値シミュレーションによって検証された。

本手法を利用して表示量を一定にする意義は以下の2つにまとめられる。

- ユーザの認知に要する負荷を一定にできる
- システムの反応性を一定にできる

第5章
提案した3次元視覚化の有効性を示すための試作システムVOGUEを構築した。システムはオブジェクト指向DBと3次元Grapherからなり、その特徴は以下のとおりである。

- 3次元グラフ表現とその立体視機能
- オブジェクト指向DBによるモデリング機能

- 対話的グラフィックス操作機能
- 図形数制御機能

第6章

VOGUEを電力制御用ソフトウェアの視覚化に適用した。まず弱い3次元の利用法の例としてソフトウェアのモジュール構造をとりあげ、モジュールの階層関係とタスク間呼びだし関係の表示を行ない、図形数制御の実用性を確認した。次に3次元の強い利用法の例として、時間軸を導入したプロセス・モニタを実現し、実際の電力制御用計算機の実行トレースを視覚化し、バグの実例を視覚的に示した。さらにメッセージ交換を基本とする将来的な分散並列システムの実行状況の視覚化を想定し、自立分散型マニピュレータにおける複数台の計算機による協調動作を取り上げ、通常状態と故障時におけるトレースを視覚化した。視覚化により、従来ボトムアップに行なっていた大規模なトレースデータの解析を、トップダウンに行なうことが可能となることを示した。また3次元の強い利用法により、プロセス(プロセッサ)間の関係と各プロセスの時間変化の同時把握が可能となるとともに、また局所情報と大局情報の同時把握が可能となった。

第7章

VOGUEをオブジェクト指向言語のクラス・ライブラリの視覚化に適用した。3次元の強い利用法によって、クラス階層関係とメソッド継承関係を分離し、かつ同時に見ることが可能となった。被験者実験の結果、メソッド探索の効率が改善された。より複雑な例としてFlavor系のオブジェクト指向言語にも対応した。その結果、複雑なデーモンの起動順番を視覚的に理解することが可能となった。また、実際の大量データ視覚化時に起こる問題点をSmalltalk-80のクラス・ライブラリの視覚化を行なうことで示し、総称関数の概念を導入したモデル定義と図形数削減手法の適用により、着目するメソッドに関連のあるクラスとメソッドだけの表示が可能となった。

第8章

視覚化システムの将来展望として、演繹データベースとの関連、人工現実感システムとの融合、制約条件が作り出す情報の形に関して述べた。さらに、第4章で開発した情報量制御手法の他の工学分野への適用例として、プログラムのソースコードの表示量制御を行なった。

謝辞

本論文は、筆者が 1988 年春から 1991 年春にかけて、東京大学大学院工学系研究科情報工学専攻博士過程在学中に行なった研究をまとめたものである。その間、指導教官石井威望教授 (東京大学工学部産業機械工学科) には終始変わらぬ熱心な御指導、御鞭撻を頂いた。

東京大学大学院工学系研究科情報工学専攻の和田英一教授、井上博允教授、田中英彦教授には、論文をまとめる上で何度となく多くの貴重な御助言を頂いた。

広瀬通孝助教授 (東京大学工学部産業機械工学科) には、ヒューマン・インタフェース一般に関して御指導頂いた。

三浦宏文教授 (東京大学工学部機械工学科) には、筆者が学部学生の頃から公私に渡り、多くの御助言、御指導頂いた。

下山勲助教授 (東京大学工学部機械工学科) には、計算機一般に関して、また私生活の面で多くの御助言を頂いた。

光石衛助教授 (東京大学工学部産業機械工学科) には、論文をまとめるにあたり多くの御助言を頂いた。

中垣好之技官、田中雅之技官には、物品の購入などで大変お世話になった。

5 年間の石井・広瀬研究室在室中には多くの先輩、同輩、後輩諸氏にお世話になった。特に、葛岡英明氏、干冬氏には常に討論の相手になって頂いた。稲村浩平君 (現在、株式会社 Canon)、三井博隆君には VOGUE のインプリメントを手伝って頂いた。その他、この間研究室に在籍していた諸氏に感謝する。

三浦純氏 (現在、大阪大学工学部助手) には研究生活、計算機についていろいろと御討論頂いた。

また、東京電力の林達郎氏、名井健氏、甘利治雄氏には快く研究環境を提供して頂

いた。また、明電舎の三神敬氏にはトレースデータを提供して頂いた。

以上の皆さんに心から謝意を表す。

最後に、この論文の完成を誰よりも待ちながら昨年11月に他界した父と、研究を続けさせてくれた母、そして妻に感謝する。

参考文献

[1] 別冊数理科学: 形・フラクタル, 1986.

[2] 数理科学: フラクタル周辺の数理, 1987.

[3] Gul A. Agha. *ACTORS: A Model of Computer Computation in Distributed Systems.* MIT Press, 1986.

[4] Amnon Aharony and Jens Feder, editors. *Fractals in Physics: Essays in honour of Benoit B. Mandelbrot.* North-Holland, 1989. Proceedings of the International Conference honouring Benoit B. Mandelbrot on his 65th birthday.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983. (邦訳) 大野 義夫 訳: データ構造とアルゴリズム, 培風館.

[6] James Ambras and Vicki O'Day. MicroScope : A Knowledge-Based Programming Environment. *IEEE Software*, 1988.

[7] 安西祐一郎. 認知科学と人工知能: インタフェースとコミュニケーション. *bit*, Vol. 18, No. 7,, 1986.

[8] 青木淳, 藤田早苗. *Smalltalk-80 Goodies Tools Applications.* 富士ゼロックス情報システムズ, 1989.

[9] 有沢誠. ソフトウェア工学. 岩波書店, 1988.

[10] M. Atkinson, F. Bancilhon, D. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proc. of the First International Conference on Deductive and Object-Oriented Databases*, 1990.

[11] Ronald M. Baecker and Aaron Marcus. *Human Factors and Typography for More Readable Programs.* ACM Press, 1990.

[12] F. T. Baker. Chief programmer team management of production programming. *IBM Systems Journal*, Vol. 11, No. 1,, 1972.

[13] R. Balzer, T. E. Cheatham, and C. Green. Software technology in the 1990's:using a new paradigm. *IEEE Computer*, 1983.

[14] M. F. Barnsley, A. Jacquin, F. Malassenet, L. Reuter, and A. D. Sloan. Harnessing chaos for image synthesis. *Computer Graphics*, Vol. 22, No. 4,, 1988.

[15] D. Barstow. Artificial intelligence and software engineering. In *Proceedings of the 9th International Conference on Software Engineering.* IEEE Software Press, 1987.

[16] S. D. Bedrosian and D. L. Jaggard. A fractal-graph approach to large networks. In *Proceedings of the IEEE*, 1987.

[17] James Bigelow. Hypertext and CASE. *IEEE Software*, 1988.

[18] Bela Bollobas. *GRAPH THEORY An Introduction Course.* Springer-Verlag, 1979. (邦訳) 斎藤, 西関 共訳: グラフ理論入門.

[19] G. Booch. *Software Engineering with Ada.* The Benjamin/Cummings Publishing, 1983.

[20] F. P.,Jr. Brooks. No silver bullet : Essence and accidents of software engineering. *IEEE Computer*, Vol. 20, No. 4, pp. 10–19, 1987.

[21] M. H. Brown. *Algorithm Animation.* MIT Press, Cambridge, MA, 1988.

[22] S. K. Card, D. Mackinlay, and G. G. Robertson. The design space of input devices. In *CHI'90 Proceedings.* ACM, 1990.

[23] Stuart K. Card and Austin Henderson, Jr. A multiple, virtual-workspace interface to support user task switching. In *CHI + GI 1987.* ACM, 1987.

[24] Ey-Chih Chow. Representing databases in frames. In *Proceedings AAAI-87.* Morgan Kaufmann, 1987.

[25] E. F. Codd. A relational model for large shared data banks. *Communication of the ACM*, Vol. 13, No. 6,, 1970.

[26] Ward Cunningham and Kent Beck. A Diagram for Object-Oriented Approach to a Large Real Time Systems. In *OOPSLA'86*, 1986.

[27] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, 1972.

[28] Susan A. Dart, Robert J. Ellison, Peter H. Feiler, and A. Nico Habermann. Software development environment. *COMPUTER*, 1987.

[29] 出原栄一. 人間と図形言語. 吉川広之 (編), コンピュータグラフィック論, chapter 3, pp. 63–117. 日科技連, 1977.

[30] 出原栄一, 吉田武夫, 渥美浩章. 図の体系 - 図的思考とその表現 -. 日科技連, 1986.

[31] K. M. Fairchild, S. E. Poltrock, and G. W. Furnas. Semnet: Three-dimensional graphic representation of large knowledge bases. In R.Guindon, editor, *Cognitive Science And Its Application For Human-Computer Interaction.* Lawrence Erlbaum Associates, 1988.

[32] Kim Fairchild, Greg Meredith, and Alan Wexelblat. The tourist artificial reality. In *CHI'89 Proceedings*, 1989.

[33] Jens Feder. *FRACTALS.* Plenum, 1988.

[34] Robert E. Filman and Daniel P. Friedman. *COODINATED COMPUTING: Tools and Techniques for Distributed Software.* McGraw-Hill, 1984. 邦訳: 雨宮, 尾内, 高橋 共訳 協調型計算システム, マグロウヒル, 1986.

[35] Franz Inc. *Allegro Common Lisp User Guide*, 1989.

[36] George W. Furnas. Generalized fisheye views. In *CHI'86*, 1986.

[37] Dedre Gentner and Donald R. Gentner. Flowing waters or teeming crowds; mental models of electricity. In Dedre Gentner and Albert L. Stevens, editors, *Mental Models.* 1983. (邦訳) 淵 一博 監修: メンタル・モデルと知識表現, 第3章 水の流れ と群れの移動: 電気のメンタル・モデル, 共立出版,1986.

[38] E. P. Glinert and S. L. Tanimoto. Pict : An interactive graphical programming environment. *IEEE Computer*, Vol. 17, No. 11, pp. 7–25, 1984.

[39] Ephraim P. Glinert. Out of flatland: Towards three-dimensional visual programming. In *1987 Fall Joint Computer Conference*. IEEE CS Press, 1987.

[40] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Xerox, 1983.

[41] L. Gould and W. Finzer. Programming by rehearsal. Technical report, Xerox Corp., 1984.

[42] P. Grogono and S. H. Nelson. *Problem Solving and Computer Programming*. Addison-Wesley, 1982.

[43] Frank G. Halasz, Thomas P. Morgan, and Randall H. Trigg. Notecards in a nutshell. In *CHI + GI 1987*. ACM, 1987.

[44] Christopher F. Herot. Spatial management of data. *ACM Transaction on Database Systems*, Vol. 5, No. 4,, 1980.

[45] Hewlett-Packard Company. *Starbase Graphics Techniques HP-UX Concepts and Tutorials Volume1*, 1988.

[46] M. Hirakawa, N. Monden, I. Yoshimoto, M. Tanaka, and T. Ichikawa. Hi-visual: A language supporting visual interaction in programming. In S. K. Chang et.al., editor, *Visual Languages*, pp. 233–259. Plenum Press, 1986.

[47] 広瀬通孝, 甘利治雄, 名井健, 石井威望, 佐々木力. 制御用言語の図形表現に関する研究. 第4回ヒューマンインタフェース・シンポジウム, 1988.

[48] 広瀬通孝, 木島竜吾, 佐藤洋一, 石井威望. シースルー型HMDを用いた仮想空間による実空間の修飾の研究. 第6回ヒューマン・インタフェース・シンポジウム論文集, 1990.

[49] D. R. Hofstadter. *Gödel,Escher,Bach: an External Golden Braid*. Basic Books, 1979. (邦訳) 野崎昭弘, 林一, 柳瀬尚紀訳, 白揚社, 東京,1985.

[50] 井田, 元吉, 大久保. Common Lisp オブジェクトシステム -CLOS とその周辺 -. 共立出版, 1989.

[51] 池井寧. 計算機組み込み型機械システムの構築手法に関する研究. PhD thesis, 東京大学大学院工学系研究科機械工学専攻, 1988.

[52] D. L. Jaggard, S. D. Bedrosian, and J. F. Dayanim. Large fractal-graph networks. 1987.

[53] P. N. Johnson-Laird. 認知科学におけるメンタルモデル. D. A. Norman, editor, 認知科学の展望, chapter 6. 産業図書, 1985.

[54] Gail E. Kaiser and Naser S. Barghouti. Database support for knowledge-based enginnering environments. *IEEE Expert*, 1988.

[55] 神沼二真, 鈴木勇. 分子を描く. 啓学出版, 1988.

[56] 上林弥彦. データベース. 昭晃堂, 1986.

[57] 金子博. フラクタル特徴とテクスチャ解析. 信学論, Vol. 70, No. 5,, 1987.

[58] Sonya E. Keene. *Object-Oriented Programming in COMMON LISP*. Addison-Wesley, 1988.

[59] B. W. Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley, 1976.

[60] Setrag N. Khoshafian and George P. Copeland. Object identity. In *OOPSLA'86*. ACM, 1986.

[61] Wolfgang Kohler. *The Task of Gestalt Psychology*. Prinston University Press, 1969.

[62] 小出昭夫. 化学 CAD におけるコンピュータグラフィックス. 情報処理, Vol. 29, No. 10,, 1988.

[63] 小池他. フラクタル, 1990. 平成 2 年度東京大学大学院石井・広瀬研究室システム工学演習資料.

[64] 小池英樹, 広瀬通孝, 石井威望. Software visualization - 3次元ソフトウェア視覚化ツールの開発 -. 第6回ヒューマン・インタフェース・シンポジウム論文集, 1990.

[65] 小池英樹, 広瀬通孝, 石井威望. Software visualization tool. 第3回人工知能学会全国大会, 1990.

[66] 国井利泰. コンピュータグラフィックス応用の発展動向. 情報処理, Vol. 29, No. 10,, 1988.

[67] 葛岡英明, 三井博隆, 広瀬通孝, 石井威望. ブラガブルなネットワーク・アプリケーション・ツールの開発. 情報処理学会ソフトウェア工学研究会, 1990.

[68] Leslie Lamport. LaTeX. Addison-Wesley, 1986.

[69] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan L. Chung, and Chaarles E. Fineman. Visualizing performance debugging. *IEEE Computer*, 1989.

[70] Peter H. Lindsay and Donald A. Norman. *Human Information Processing -An Introduction to Psychology 2nd Ed.* Academic Press, 1977. (邦訳) 中溝, 箱田, 近藤 共訳: 情報処理心理学入門, サイエンス社,1984.

[71] David Maier and Jacob Stein. Development of an Object-Oriented DBMS. In *OOPSLA'86*, 1986.

[72] B. B. Mandelbrot. *The Fractal Geometry of Nature.* W.H.Freeman and Company, 1977. (邦訳) 広中平佑 訳: フラクタル幾何学, 日経サイエンス社,1982.

[73] 増永良文. マルチメディアデータベース総論. 情報処理, Vol. 28, No. 6,, 1987.

[74] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, Vol. 21, No. 4,, 1989.

[75] Bertrand Meyer. Genericity versus inheritance. In *OOPSLA'86.* ACM, 1986.

[76] Norman Meyrowitz. Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework. In *OOPSLA'86*, 1986.

[77] 宮崎清孝. 理解と視点 - 概念理解の場合. 佐伯 (編), 認知心理学講座 第3巻 推論と理解, chapter 2. 東京大学出版会, 1982.

[78] M. Moriconi and D. F. Hare. Visualizing Program Designs Through PegaSys. *IEEE Computer*, Vol. 18, No. 8,, 1985.

[79] 守屋慎次. ユーザインタフェース技法. 情報処理, Vol. 29, No. 10,, 1988.

[80] Brad A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, Vol. 17, No. 3,, 1983.

[81] Brad A. Myers. Visual programming, programming by example and program visualization: A taxonomy. In *Proc. CHI '86: Human Factors in Computing Systems*, pp. 59–66. ACM, 1986.

[82] Brad A. Myers. The state of art in visual programming and program visualization. Technical report, CMU, 1988.

[83] 名井健, 甘利治雄, 林達郎, 広瀬通孝, 石井威望, 小池英樹: 視覚化表現を用いた制御用ソフトウェアの設計. 第6回ヒューマン・インタフェース・シンポジウム論文集, 1990.

[84] Ulric Neisser. *Cognitive Science*. Prenntice-Hall, 1967. (邦訳) 大羽 訳: 認知心理学, 誠信書房,1981.

[85] T. V. Papathomas, J. A. Schiavone, and B. Julesz. Stereo animation for very large data bases: Case study - meteorology. *IEEE Computer Graphics and Application*, 1987.

[86] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communication of the ACM*, Vol. 15,, 1972.

[87] D. L. Parnas. Active design reviews:principles and practices. In *Proceedings of the 8th International Conference on Software Engineering*, 1985.

[88] George Raeder. A survey of current graphical programming techniques. *IEEE Computer*, 1985.

[89] J. Ramanathan and R. L. Hartung. A Generic Iconic Tool for Viewing Databases. *IEEE Software*, Vol. 6, No. 5,, 1989.

[90] Marcello G. Reggiani and Franco E. Marchetti. A proposed method for representing hierarchies. *IEEE Trans. of systems,man, and cybernnetics*, Vol. 18, No. 1,, 1988.

[91] Edward M. Reingold and John S. Tilford. Tidier drawings of trees. *IEEE Trans. on Software Engineering*, Vol. 7, No. 2,, 1981.

[92] David E. Rumelhart, James L. McClelland, and The PDP Research Group. *Parallel Distributed Processing*. MIT Press, 1986.

[93] R. N. Shepard and J. Metzler. Mental rotation of three-dimensional objects. *Science*, Vol. 171,, 1971.

[94] N. C. Shu. *Visual Programming*. Van Nostrand Reinhold, New York, 1988.

[95] Andrea H. Skarra and Stanley B. Zdonik. The management of changing types in an object-oriented database. In *OOPSLA'86*, 1986.

[96] S.K.Chang. Visual languages: A tutorial and survey. *IEEE Software*, Vol. 4, No. 1,, 1988.

[97] R. B. Smith. The alternate reality kit: An animated environment for creating interactive simulations. In *1986 Workshop on Visual Languages*. IEEE, 1986.

[98] Kathryn T. Spoehr and Stephen W. Lehmkuhle. *Visual Information Processing*. W.H.Freeman and Company, 1982.

[99] Guy L. Steele Jr. *Common Lisp The Language*. DEC, 1984. (邦訳) 後藤 英一 監訳 Common Lisp 言語仕様書.

[100] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986. (邦訳) 斉藤 信男訳: プログラミング言語 C++, トッパン,1988.

[101] 杉山公造. ヒューマン・インタフェースとしての図形言語 III : 認知的基準によるグラフ描画. 第3回ヒューマン・インタフェース・シンポジウム, 1987.

[102] 鈴木則久 (編). オブジェクト指向 解説と WOOC'85 からの論文. 共立出版, 1985.

[103] 高橋義造 (編). 並列処理機構. 丸善株式会社, 1989.

[104] 高野陽太郎. 傾いた図形の謎. 東京大学出版会, 1987.

[105] 高安秀樹. フラクタル. 朝倉書店, 1986.

[106] 高安秀樹. フラクタル科学. 朝倉書店, 1987.

[107] Roberto Tamassia, Giuseppe Di Battista, and Calo Batini. automatic graph drawing and readability of diagrams. *IEEE Trans. of systems,man, and cybernnetics*, Vol. 19, No. 1,, 1988.

[108] Steven L. Tanimoto and Ephraim P. Glinert. Designing iconic programming systems: Representation and learnability. In *1986 Workshop on Visual Languages*. IEEE CS Press, 1986.

[109] W. Teitelman. A display oriented programmer's assistant. *International Journal of Man-Machine Studies*, Vol. 11,, 1979.

[110] W. Teitelman. A Tour Through Cedar. *IEEE Transactions on Software Engineering*, Vol. 11, No. 3,, 1985.

[111] Ryuji Tokunaga. *Chaos and Bifurcations in Autonomous Electrical Circuits.* PhD thesis, Waseda University Department of Electrical Engineering.

[112] 鳥居修晃. 現代基礎心理学第 3 巻. 東京大学出版会, 1982.

[113] 上谷晃弘 (編). 統合化プログラミング環境 –Smalltalk-80 と Interlisp-D–. 丸善株式会社, 1987.

[114] Steven R. Vegdahl. Moving structures between Smalltalk images. In *OOPSLA '86*, 1986.

[115] Tamas Vicsek. *Fractal growth phenomena.* World Scientific, 1989.

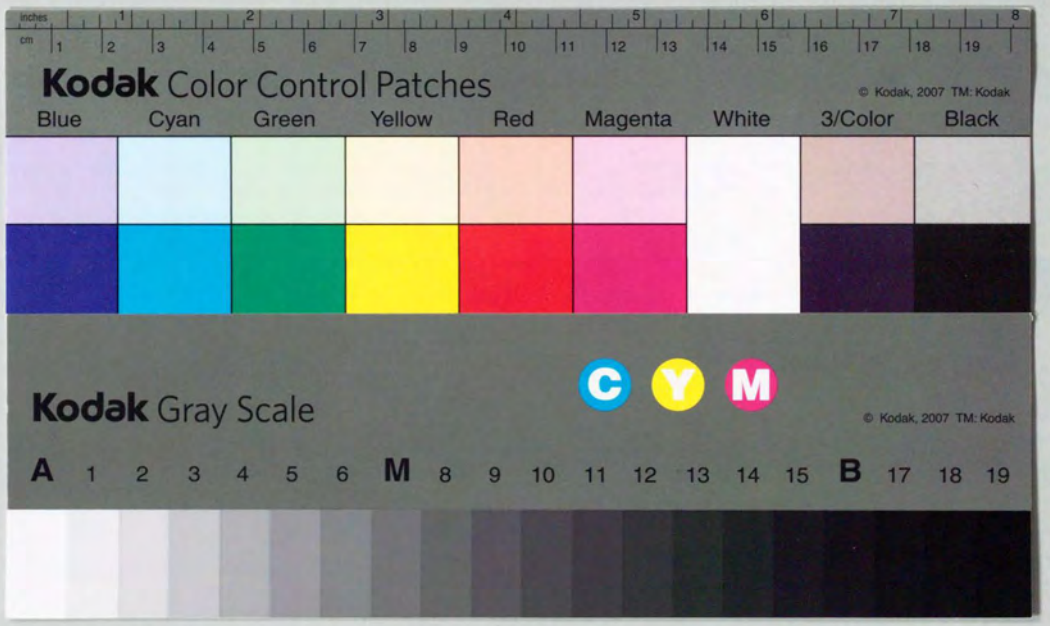
[116] A. I. Wasserman. *Tutorial:Software Development Environments.* IEEE Computer Society Press, 1981.

[117] Charles Wetherell and Alfred Shannon. Tidy drawings of trees. *IEEE Trans. on Software Engineering*, Vol. 5, No. 5,, 1979.

[118] N. Wirth. Program development by stepwise refinement. *Communication of the ACM*, Vol. 14, No. 4,, 1971.

[119] N. Wirth. Modula:a language for modular multiprogramming. *Software - Practice and Experience*, Vol. 7, No. 1,, 1977.

[120] E. Yourdon. *Structured Walkthroughs.* Prentice-Hall, 1979.


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19