ビジュアルシステム生成系への
レイアウト制約の導入

工学研究科

筑波大学

2000 年 7 月

丁　錫泰

要旨

本論文では、ビジュアルシステム生成系にレイアウト制約を導入したビジュアルシステム生成系の研究について述べる。本システムでは、ビジュアルシステムの文法の仕様を与えることにより、図形を解釈しながらインタラクティブに図形をバランスよくレイアウトすることができるビジュアルシステムを生成する。また、本システムでは、図形のレイアウトの指定にレイアウト制約を使用する。

我々は、レイアウト制約として軟かいレイアウト制約と硬いレイアウト制約の二種類を提案した。軟かいレイアウト制約は、図形の全体を自動描画アルゴリズムに従って分りやすくレイアウトする制約である。軟かいレイアウト制約として、スプリングモデル制約、マグネティックスプリングモデル制約、木構造制約などを導入した。ここで、スプリングモデル制約は無向グラフのレイアウトを行う場合に用いる。マグネティックスプリングモデル制約は、エッジの方向を考えて有向グラフのレイアウトを行いたいときに用いる。また、木構造制約は、グラフを木構造にレイアウトする場合に用いる。硬いレイアウト制約は、特定の図形の座標や図形間の距離などを具体的に与える場合に用いる制約である。

本論文の新規性は、軟かいレイアウト制約を CMG （Constraint Multiset Grammars）の生成規則として扱うことにより、図形をグローバルにレイアウトすることが可能になった点である。また、軟かいレイアウト制約と硬いレイアウト制約を融合することにより、適用できるビジュアルシステムの応用範囲を広げることができた。さらに、ビジュアルシステム生成系にレイアウト制約を追加することにより、図形のパーシング中に自動描画することで、図形をよりインタラクティブに処理することができた点である。

我々は、レイアウト制約を導入したビジュアルシステム生成系「Rainbow」を開発した。「Rainbow」によるビジュアルシステム作成の例として、データベース分野で実世界のデータ構造を記述するのに用いられる「E-R ダイアグラム」、オブジェクト指向に基づくソフトウェア設計に用いられる「オブジェクト図」、会社などの仕組みを表すのに用いられる「組織図」、親族の関係を表すのに用いられる「家系図」などを示した。

目次

要旨 1

1 序論 6

1.1 研究の背景 6
1.2 研究の動機 7
1.3 研究の目的 8

2 レイアウト制約 9

2.1 軟かいレイアウト制約 9
2.1.1 CMG 9
2.1.2 グラフ描画アルゴリズム 12
2.1.3 軟かいレイアウト制約とは 13
2.1.4 軟かいレイアウト制約の種類 14
2.1.5 軟かいレイアウト制約の扱い方 14
2.2 硬いレイアウト制約 19
2.2.1 硬いレイアウト制約の種類 19
2.2.2 硬いレイアウト制約の扱い方 20
2.2.3 通常の制約と硬いレイアウト制約の違い 21
2.2.4 空間パーサとレイアウト制約との関係 22

3 システム「Rainbow」 24

3.1 「Rainbow」の概要 24
3.2 「Rainbow」の構造 29
3.3 「Rainbow」の解釈アルゴリズム 31

3.4 レイアウト制約モジュール 33
3.4.1 軟らかいレイアウト制約モジュール 33
3.4.2 硬いレイアウト制約モジュール 33
3.4.3 「Rainbow」と恵比寿の比較 34

4 「Rainbow」の適用例 35
4.1 スプリングモデル制約 35
4.1.1 スプリングモデル 35
4.1.2 スプリングモデル制約の例 36
4.2 マグネティックスプリングモデル制約 38
4.2.1 マグネティックスプリングモデル 38
4.2.2 マグネティックスプリングモデル制約の例 39
4.3 木構造制約 42
4.3.1 木構造 42
4.3.2 木構造制約の例 44
4.3.3 木構造制約と硬いレイアウト制約を混ぜた例 47

5 「Rainbow」の評価実験 52
5.1 実験環境 52
5.2 評価実験 1 53
5.3 評価実験 2 56

6 関連研究 58
6.1 空間パーサ生成系の関連研究 58
6.2 レイアウトシステムの関連研究 59

7 結論 60

参考文献 62

謝辞 66

図一覧

2.1 リスト構造 10
2.2 硬いレイアウト制約の例 21

3.1 「Rainbow」の図形エディタ 25
3.2 「Rainbow」の CMG 入力ウィンドウ (1) 26
3.3 「Rainbow」の CMG 入力ウィンドウ (2) 27
3.4 レイアウト前後のリスト構造 28
3.5 リストの軟かいレイアウト CMG 入力ウィンドウ (1) 29
3.6 リストの軟かいレイアウト CMG 入力ウィンドウ (2) 29
3.7 「Rainbow」の構成図 30
3.8 生成規則の内部表現 30
3.9 解析アルゴリズム 32

4.1 スプリングモデルによるグラフのレイアウト 36
4.2 レイアウト前の E-R ダイアグラム 37
4.3 レイアウト後の E-R ダイアグラム 38
4.4 マグネティックスプリングモデルによるグラフのレイアウト 38
4.5 オブジェクト図の構成要素 40
4.6 レイアウト前のオブジェクト図 40
4.7 レイアウト後のオブジェクト図 41
4.8 木構造のレイアウト規則モジュールの内部表現 43
4.9 「Rainbow」の図形エディタ 44
4.10 ノードの CMG 入力ウィンドウ (1) 45
4.11 ノードの CMG 入力ウィンドウ (2) 46

4.12 エッジの CMG 入力ウィンドウ 47
4.13 レイアウト前の組織図 48
4.14 組織図の軟かいレイアウト CMG 入力ウィンドウ (1) 49
4.15 組織図の軟かいレイアウト CMG 入力ウィンドウ (2) 49
4.16 レイアウト後の組織図 50
4.17 レイアウト前の家系図 50
4.18 レイアウト後の家系図 51

5.1 E-R ダイアグラム 53
5.2 家系図 54
5.3 E-R ダイアグラムの実行結果 56
5.4 家系図の実行結果 57

第 1 章

序論

1.1 研究の背景

図形は情報の表現や伝達さらには思考の道具として広い範囲で用いられている。図形の中で図形要素や図形要素の配置規則がはっきりしており、文章や数式と同じように、意味が一義的に導き出せるものを特に図形言語と呼ぶ [1]。図形言語は、テキスト言語より情報を理解しやすくする。情報を表すとき、テキスト言語では文字を一次元に並べるが、図形言語では図形や立体を二次元あるいは三次元に配置することで情報を視覚的に表すことができる。こうした図形言語は、図形要素間になり立っている関係を表す空間的な文法を持っていると考えることができる。

空間パーサ生成系とは、図形の空間的な文法を定義することで図形の空間パーサを生成するシステムのことである [2] [3] [4] [5] [6]。

空間パーサ生成系についての研究として、 SPARGEN[2] や Penguins[3] [4] などの研究がある。 Golin らにより提案されている SPARGEN は、 OOPLG（Object-Oriented Picture Layout Grammars）を用いて図形言語の文法を定義することで空間パーサを生成するシステムである。 OOPLG では、図形の属性や制約を C^{++} を用いて定義している。また、 Marriot らの Penguins では、図形の文法として Constraint Multiset Grammars （CMG） [3] [7] を用いて空間パーサを生成している。

しかしながら、これらの空間パーサ生成系は、テキストを用いて図形言語の文法を定義するので、ユーザは文法を理解している必要があり、一般のユーザにとって使いやすいものとはいえないという問題があった。

そこで、我々が研究を行っている恵比寿 [5] [6] [8] では、ユーザが入力した図形

を用いて大まかな図形の CMG の文法を自動的に生成するようにしている。そのあとユーザは、生成された文法を見ながら、制約の追加または削除などをおこない修正する。また、 VIC[9] では視覚的な制約入力インターフェイスを恵比寿上に実装し、テキスト編集を行わずに CMG の文法を生成している。

図形を処理するビジュアルシステムは、空間パーサを用いて図形を解釈し、その図形を描き変えることが多い。ここで、図形を描き変えるということは、図形を生成、削除、移動したり、図形の属性の値（色や文字列の値、線の太さなど）を変更するアクションである。

図形が描き変えられるとき、図形の生成により図形の数が多くなって重なることや図形の削除により図形間の位置関係が分かりにくくなることなどの問題が生じる。この問題の解決方法としては、ユーザが直接図形をレイアウトする方法がある。しかし、図形を処理するシステムはもっとインタラクティブであるべきだと考える。

本研究では、ビジュアルシステム生成系にレイアウト制約を扱えるようにし、図形の一部または全体を解釈しながらバランスよくレイアウトすることができるビジュアルシステムの研究を行った [10]。その研究に基づいて、我々はビジュアルシステム生成系「Rainbow」を開発した。「Rainbow」は、 我々がこれまで研究を行ってきた恵比寿にレイアウト制約を追加することにより実現している。

1.2 研究の動機

我々は、オブジェクト指向に基づくソフトウェア開発方法論である OMT （Object Modeling Technique） [11] のモデルから実行可能なオブジェクト指向コードに変換を行うための新しい手法について研究してきた。特に、設計と実装フェーズにおいて、オブジェクトモデル中のクラスの振舞いを分かるため最も重要な動的モデルの仕様から実行可能な Java コードを生成する方法について研究を行った [13] [14] [15]。

また、クラスの振舞いを表す動的モデルの活動（activity）、動作（action）に「アイコン変形（icon transformation）」を定義し、それからシステムをアニメーションさせるコードを生成する方法を研究した [16] [17]。

これらの研究で用いられたオブジェクトモデル、動的モデルを表現するのに使わ

れる図形表記のオブジェクト図、状態遷移図は、分析フェーズから設計フェーズ、実装フェーズまでを続ぎ目なく同一の表記で表すことにより、ある開発フェーズで追加された情報を、次のフェーズで失ったり翻訳し直したりする必要がなくなる。また、これらの図形はシステムを開発する複数の開発者により利用することが可能であり、開発者間のコミュニケーションの道具になる。

今まで研究されている空間パーサでは、図形の一部または全体を解釈しながら分かりやすくレイアウトすることができない。ビジュアルシステム生成系にレイアウト制約を導入することにより、これらの問題を扱うことが可能ではないかと考えたのが本研究の動機である。

1.3 研究の目的

本研究の目的は、ビジュアルシステム生成系にレイアウト制約を導入し、扱えるビジュアルシステムの範囲を広げることである。

「Rainbow」を用いることにより、ソフトウェア開発において重要視される各種の設計図、工程図、データの流れ図、回路図、E-R ダイアグラム、組織図、家系図、および、テレビドラマの登場人物の関係を表す図などの様々な図形をもっとインタラクティブに扱うビジュアルシステムを作ることが可能になる。すなわち、空間パーサにレイアウト制約を追加することにより、パーシング中に図形を自動描画することで図形をよりインタラクティブに処理するビジュアルシステムを作ることが可能である。

特に、従来のオブジェクト指向 CASE ツール [18] [19] は、モデルを表現するのに使われる図形表記を描くための図形エディタを備えている。しかしながら、それらの図形エディタでは、図形表記を構成する基本部品をユーザがすべて配置するといった操作環境となっているため、ユーザが分かりやすいレイアウトを考慮しながら配置していなければならない。ここで、「Rainbow」はインタラクティブな CASE ツールを作るための重要な要素技術の一つを提供すると考えられる。

第 2 章

レイアウト制約

図形による視覚的表現は直感的に理解が容易であり情報伝達の手段として有効であるが、複雑な構造や関係を持つ図形の場合は可読性や美しさを考慮してレイアウトしないと図形による視覚的表現のメリットを十分に活かすことができない。

レイアウト制約はユーザにより図形の文法にレイアウトの規則として定義され、図形がその文法により解釈された後、もともとの図形要素間の複雑な構造や関係を分かりやすくバランス良く表す位置関係を作り出す制約である。

2.1 軟かいレイアウト制約

我々は、レイアウト制約として軟かいレイアウト制約と硬いレイアウト制約の 2 種類を考える。

軟かいレイアウト制約は、図形の全体を自動描画アルゴリズムに従ってバランスよく分りやすくレイアウトする制約である。一方、硬いレイアウト制約は、図形の座標や図形間の距離などの制約を具体的に与えたい場合に用いる。

軟かいレイアウト制約と硬いレイアウト制約について述べる前に、図形言語の文法を定義するために用いられる Constraint Multiset Grammars（CMG）、グラフ描画アルゴリズムに関して説明する。

2.1.1 CMG

CMG[3] [7] は終端シンボルの集合、非終端シンボルの集合、開始シンボル、および、生成規則の集合から構成される。すべての終端シンボルと非終端シンボルは、

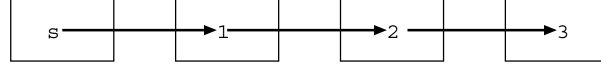

図 2.1: リスト構造

色、大きさ、位置などの属性を持っている。生成規則はトークン（終端シンボルもしくは非終端シンボルのインスタンス）のマルチセットとそれらの属性間の関係を保つ制約により構成される。ここで、マルチセットを用いる理由は、同じ種類のトークンでもそれぞれを別々のトークンとして扱うためである。本論文では右辺から左辺への書き換えを生成と呼ぶ。生成規則は次のように定義される。

$$\begin{aligned} T(\vec{x}) \quad ::= \quad & T_1(\vec{x_1}), \cdots, T_n(\vec{x_n}) \ where \\ & exists\ T'_1(\vec{x'_1}), \cdots, T'_m(\vec{x'_m}) \\ & where\ C(\vec{x_1}, \cdots, \vec{x_n}, \vec{x'_1}, \cdots, \vec{x'_m})\ and \\ & \vec{x} \ = \ F(\vec{x_1}, \cdots, \vec{x_n}, \vec{x'_1}, \cdots, \vec{x'_m}) \end{aligned}$$

すなわち、トークンのマルチセット T_1, …, T_n （`normal` の構成要素）、T'_1, …, T'_m （`exist` の構成要素）の属性が制約 C を満たす場合、T_1, …, T_n が非終端シンボル $T(\vec{x})$ に書き換えられることを意味している。制約は、生成規則の構成要素になっているシンボルの属性間に成り立っている関係を表すものであり、生成規則が適用されると、構成要素の属性間に制約が課せられ、それらの関係を保存したまま図形の編集を可能にするのに用いられる。 exist の構成要素は、定義したい $T(\vec{x})$ を認識するために、存在する必要がある構成要素である[1]。 F は、構成要素の属性 $\vec{x_1}, \cdots, \vec{x_n}$ と $\vec{x'_1}, \cdots, \vec{x'_m}$ を引数とする関数であり、定義中の非終端シンボルの属性に値を与えることを定義している。

図 2.1のようなリスト構造を考えてみる。リスト構造の文法を定義するためには、次のように「再帰的に定義する生成規則」が必要になる。

生成規則 1 四角の中心にラベルとして `s` が書いてあるものをリストとする。

[1]その他に CMG は構成要素として `not_exist`、 `all` などを持っている。詳しくは文献 [3] [5] [7] を参照。

生成規則 2 一つのリストが矢印によって四角につながれ、その四角の中心にラベルとして数字が書いてあるものをリストとする。

これらを CMG で記述すると次のようになる。

```
 1:list(point mid) ::= R:rectangle, T:text
 2:   where (
 3:      R.mid  == T.mid  &&
 4:      T.text == ‘‘s’’
 5:   ) {
 6:   mid = R.mid;
 7: }
 8:
 9:list(point mid) ::= R:rectangle, T:text,
10:           L:line, LL:list
11:   where (
12:      R.mid   == T.mid   &&
13:      R.mid   == L.end   &&
14:      LL.mid  == L.start
15:   ) {
16:   mid = R.mid;
17: }
```

生成規則 1 は図 2.1のラベル s の四角をリストとして解釈するための生成規則で、1 行目から 7 行目までが CMG の定義である。1 行目は四角（R）とテキスト（T）で構成されている非終端シンボル「リスト（list）」を定義している。リストは、属性として中心（mid）を持つ。2、3、4、5 行目は、リストの構成要素間の制約を定義している。3 行目は、四角の中心（R.mid）とテキストの中心（T.mid）が等しいという制約を表す。4 行目は、テキストの文字列（T.text）が「s」であることを表している。この二つの制約を満たすときに 6 行目が行われる。6 行目はリストの中心として四角の中心の値を代入することを表している。

生成規則 2 はラベルが付いている四角に一つのリストがつながっているものをリストとして解釈するための生成規則で、 9 行目から 17 行目までが CMG の定義である。 13 行目は四角の中心と直線の終点（`L.end`）が一致する制約、 14 行目はリストの中心（`LL.mid`）と直線の始点（`L.start`）が一致する制約を意味している。

2.1.2　グラフ描画アルゴリズム

図による視覚的な表現はインターフェースとして最も基本的なものの一つであり、使われる範囲も広く、簡便で親しみやすいという特徴を持っている。よって設計図、家系図、論理回路図など日常的にも多く利用されている。なかでも我々が普段、研究や仕事に扱う事の多い、フローチャート、組織図、システム構成図などは実体をノード、実体間の関係をエッジとしたグラフとして表現される。またこういったグラフはシステム工学、情報工学、ソフトウェア工学など様々な分野において、基礎的なモデルとして広く使われている。最近ではコンピュータの発達にともない、グラフを視覚的に表示したり、操作したりする事が強く求められるようになってきている。そのためこのような図の自動レイアウトを行う、グラフ描画アルゴリズムが数多く提案されている。

グラフ描画アルゴリズムが対象とするグラフは、その図的性質によって様々である。それゆえグラフの種類によって美的基準もまちまちである。そこで各クラスの特徴を考慮した描画法がグラフの種類ごとに開発されている。

グラフでいう美的基準とは、描画規約と描画規則のことである [1]。描画規約はノードとエッジに関する基本的約束であり、描画の際に必ず満たされる制約である。描画規則は “良い” 描画の基準となるものである。ただどのようなグラフが “良い” かは個人により変化し、主観による部分が大きい。しかし、グラフの構成はノードとエッジというように、極度に単純化できるため、細かい違いはあるものの、グラフの性質からくる、共通で、基本的ないくつかの良い描画の基準を識別する事ができる。それらの描画規則の例をいくつか挙げる。

- エッジの折れ点数を最少とする
- エッジの交差数を最少にする
- 対称性 (がある場合) を顕示する

- ノードとエッジの配置や配線の密度を一様化する
- 子ノードを対称に配置する
- 階層構造を垂直あるいは水平に顕示する

グラフ描画アルゴリズムとは一般的に、対象とするグラフの特長から優先度の高い順に、これらの描画規則を取り出し、最適基準または制約条件とする。そして描画規約を最適化問題のゴールとして、複数のステップにおいて順次最適化問題を解くことにより、多くの描画規則を満たしていくものである。しかし、木などの交差の無いグラフを除くと、規則を満たせる効率的な方法が無いことが多く、最適解を得ることは困難なことが多い。したがって、ヒューリスティクスを用いた種々の発見的方法によるアルゴリズムも開発されてきている。

グラフ描画アルゴリズムが対象とするグラフは、木、有向グラフ、無向グラフ、複合グラフの各クラスに分類される。

木は、もともと平面グラフなのでエッジの交差なしにレイアウトされる。また、木は組織図のような階層構造を表すのによく用いられる。有向グラフは、各エッジの方向を一定方向の流れとして階層的に、または、各エッジを区別してレイアウトされることが多い。無向グラフは、交差するエッジがないように平面に描くことができる。また、無向グラフはグラフ理論、グラフアルゴリズム、さらにグラフ描画法において最もよく研究されている基礎的かつ重要なクラスである。木、有向グラフ、無向グラフは、ノード間の隣接関係を表すグラフに対して、複合グラフは隣接関係と包含関係を表すグラフである。複合グラフは、様々な分野において人間の思考を助長するための道具として用いられる。

2.1.3 軟かいレイアウト制約とは

軟かいレイアウト制約とは、図形の全体を自動描画アルゴリズムに従ってバランスよくレイアウトする制約である。

ユーザにより図形の特定な部分がレイアウトされても図形の全体を考えるとバランスよく、または、見やすくレイアウトされてない場合が多い。レイアウトでは、図形の特定な部分をレイアウトすることより、軟かいレイアウト制約のように図形の全体を把握しやすく、また、見やすくレイアウトすることが大事である。

2.1.4 軟かいレイアウト制約の種類

グラフ描画アルゴリズムでは、木やグラフの描画を扱う。無向グラフを描画するのに最も基本となるアルゴリズムとしてスプリングモデル [20] がある。また、このスプリングモデルをもとにエッジの方向を考えることにより、有向グラフの描画を行うマグネティックスプリングモデル [21] がある。さらに、木を描画するのに現在最も優れていると考えられる Walker の一般木描画アルゴリズム [22] が存在する [1]。

我々は、これらの木やグラフをインタラクティブにレイアウトするための軟かいレイアウト制約として、スプリングモデル制約、マグネティックスプリングモデル制約、木構造制約を考える。

軟らかいレイアウト制約の名前として、スプリングモデル制約は `spring`、マグネティックスプリングモデル制約は `magnetic`、木構造制約は `treeStructure` を用いる。

2.1.5 軟かいレイアウト制約の扱い方

軟かいレイアウト制約を CMG に基づいて考えると、 **CMG の制約として扱う方法**と **CMG の生成規則として扱う方法**が考えられる。

軟かいレイアウト制約を **CMG の制約**として扱う方法では、生成規則に以下のように軟かいレイアウト制約を指定する。この生成規則には、 CMG の生成規則に $softConstraintName$ が追加されている。

$$
\begin{array}{rcl}
T(\vec{x}) & ::= & T_1(\vec{x_1}), \cdots, T_n(\vec{x_n})\ where \\
 & & exists\ T'_1(\vec{x'_1}), \cdots, T'_m(\vec{x'_m}) \\
 & & where\ C(\vec{x_1}, \cdots, \vec{x_n}, \vec{x'_1}, \cdots, \vec{x'_m})\ and \\
 & & softConstraintName\ and \\
 & & \vec{x}\ =\ F(\vec{x_1}, \cdots, \vec{x_n}, \vec{x'_1}, \cdots, \vec{x'_m})
\end{array}
$$

この生成規則の意味は、 `normal` の構成要素や `exist` の構成要素のマルチセットの属性が制約 C を満たす場合、 `normal` の構成要素が非終端シンボル $T(\vec{x})$ に書き

換えられることを表す。さらに、構成要素に指定された軟らかいレイアウト制約の名前（$softConstraintName$）に従ってレイアウトが与えられることを表している。

しかしながら、よく考えてみると軟かいレイアウト制約を **CMG の制約**として扱う方法では、図形の全体のレイアウトはできないことが分かる。

その理由は次の通りである。グラフ構造のような図形の文法は、例としてあげられたリスト構造のように「再帰的に定義する生成規則」により定義される。図形にこの生成規則が再帰的に適用されると、まず基本的な図形（すなわち円、四角、直線、テキストなど）を組み合わせて非終端シンボルを作る。それから、その非終端シンボルと他のシンボルを組み合わせて非終端シンボルを作ることを繰り返す。我々は、このような非終端シンボルを「再帰的に生成される非終端シンボル」と呼ぶ。また、我々は「再帰的に生成される非終端シンボル」を繰り返し作ることを低い階層を持つ非終端シンボルから高い階層を持つ非終端シンボルを生成して行くことであると考える。例えば、リスト構造を例にすると、生成規則 1 により、四角とテキストを組み合わせて非終端シンボル `list` を生成する。生成規則 2 により、 `list`、四角、テキスト、直線を組み合わせて `list` を生成する。生成規則 2 が繰り返し適用されることにより、図形の全体を `list` として認識する。軟かいレイアウト制約を CMG の制約として扱うと、「再帰的に生成される非終端シンボル」が生成されるたびに軟かいレイアウト制約をその非終端シンボルの構成要素に与える。そうすると、低い階層を持つ非終端シンボルから軟かいレイアウト制約が与えられて行くので、図形の一部のレイアウトは可能だが、図形の全体をバランスよくレイアウトすることはできない。

そこで次に、軟かいレイアウト制約を **CMG の生成規則**として扱う方法について考える。軟かいレイアウト制約を CMG の生成規則（軟らかいレイアウトの生成規則と呼ぶ）として扱うと、生成規則は図形の全体に適用されるので、図形の全体をバランス良くレイアウトすることが可能であると思われる。

軟かいレイアウト制約とは、図形の全体をグラフ描画アルゴリズムに従ってレイアウトする制約であるので、図形の全体を `node` と `edge` で構成するグラフ構造に対応させて制約を与える。

また、 `node` や `edge` の構成要素を扱うためには、図形の中に `node` の構成要素や

edge の構成要素にしたい非終端シンボルを生成する生成規則が必要になる。その他に、図形のパーシングを行うため、 node の構成要素や edge の構成要素を用いて、図形を再帰的に定義する生成規則が必要になる。

まず、リスト構造[2]を例にして node や edge の構成要素を定義するための生成規則を定義する。

生成規則 1 四角の中心にラベルとしてテキストが書いてあるものをノードとする。

```
1: lNode(point mid, string text) ::=
2:              R:rectangle, T:text
3:   where (
4:      R.mid == T.mid
5:   ) {
6:   mid  = R.mid;
7:   text = T.text;
8: }
```

生成規則 2 ノード間を結ぶ直線をエッジとする。

```
1: lEdge(point start, point end) ::= L:line
2:   where ( exist N1:lNode, N2:lNode
3:      where (
4:         L.start == N1.mid  &&
5:         L.end   == N2.mid
6:   ) {
7:   start = L.start;
8:   end   = L.end;
9: }
```

次は、 lNode や lEdge を用いてリスト構造を「再帰的に定義する生成規則」を定義する。

[2]リスト構造は木構造や他のグラフ構造より単純な構造を持っているので、軟かいレイアウト制約として扱う必要がないが、説明を分かりやすくするために論文の中でリスト構造の例題を扱う。

生成規則 3 リスト構造の生成規則 1 と同じであるが、構成要素がノードである。

```
1:list(point mid, string text) ::= N:lNode
2:   where (
3:      N.text == ‘‘s’’
4:   ) {
5:   mid  = N.mid;
6:   text = N.text;
7: }
```

生成規則 4 リスト構造の生成規則 2 と同じであるが、構成要素がリスト、エッジ、ノードである。

```
1:list(point mid, string text) ::=
2:           L:list, E:lEdge, N:lNode
3:   where (
4:      E.start == L.mid  &&
5:      E.end   == N.mid
6:   ) {
7:   mid  = N.mid;
8:   text = N.text;
9: }
```

我々は、軟らかいレイアウトの生成規則を以下のように定義する。

$$
\begin{aligned}
S(\vec{x}) \;\; ::= \;\; & nodes\ S_1, \cdots, S_n \\
& edges\ S'_1, \cdots, S'_m \\
& where\ SC\ and\ GS(\vec{x_1}, \cdots, \vec{x_n})\ and \\
& \vec{x} \;=\; F(\vec{x_1}, \cdots, \vec{x_n})
\end{aligned}
$$

ここで、 S は非終端シンボルの名前、 S_1，…， S_n は $node$ の構成要素の名前、 S'_1，…， S'_m は $edge$ の構成要素の名前、 SC は軟らかいレイアウト制約の名前、

GS は再帰的に生成される非終端シンボルの名前を示している。ここで、軟らかいレイアウト制約の名前として、 spring、 magnetic、 treeStructure が用いられる。

軟らかいレイアウトの生成規則は、通常の生成規則と異なり、図形の全体に存在する node の構成要素 S_1，⋯， S_n のマルチセットと edge の構成要素 S'_1，⋯， S'_m のマルチセットを GS の構成要素から求めて非終端シンボル S として認識し、指定された軟らかいレイアウト制約の名前 SC に従ってレイアウトを行うことを意味する。

各非終端シンボルは、自分の構成要素の情報を持っているので、軟かいレイアウトの生成規則が適用される時に、 GS の構成要素が GS を持たなくなるまで繰り返し検査し、 node や edge の構成要素のマルチセットを動的に求めて、 SC に従ってレイアウトを行うことができる。このため、図形エディタに node や edge の構成要素になっている非終端シンボルが追加・削除されても、きちんとレイアウトを行うことができる。

また、複数の図形があるときに、それぞれの図形に軟らかいレイアウトの生成規則を与え、きちんとレイアウトを行うことができる。複数の図形がパーシングされると、複数の図形と同じ数の GS が生成される。それぞれの GS の構成要素から、それぞれの図形に対する node の構成要素や edge の構成要素のマルチセットを求め、軟らかいレイアウトの生成規則を適用すれば可能である。

リスト構造の軟らかいレイアウトの生成規則は、以下のように定義される。

```
1: layoutList(point mid) ::= nodes:lNode,
2:                           edges:lEdge
3:   where (
4:      listStructure  &&
5:      list
6:   ) {
7:   mid = list.mid;
8: }
```

ここで、 4 行目の listStructure は軟かいレイアウト制約の種類の名前である

リスト構造制約（SC）、5 行目の `list` は「再帰的に生成される非終端シンボル」の名前（GS）を示している。

2.2 硬いレイアウト制約

硬いレイアウト制約とは、図形の一部をローカルにレイアウトする制約である。特に、図形の座標や図形間の距離などの制約を具体的に与えたい場合に用いる。

2.2.1 硬いレイアウト制約の種類

図形の座標を一致させて図形を描画する制約は具体的に次のように記述する。

`layout_eq` 変数 1 変数 2

ここで、変数 1 と変数 2 は終端シンボルもしくは非終端シンボルの属性を示している。変数 1 と変数 2 の型としては integer、 point を用いることができる。変数 2 の値として変数 1 の値を代入して、変数 2 を属性として持つ図形を変った位置に描画する。例えば、図形の構成要素としてノード `N1` と `N2` があるとする。 `layout_eq N1.mid_y N2.mid_y` は、 `N1` と `N2` の構成要素が解釈された後、 `N2` の構成要素の属性 `mid_y` （中心の y 座標）の値を `N1` の構成要素の属性 `mid_y` の値と等しくして `N2` の構成要素を描画する。

図形間の距離を具体的に与えて図形を描画する制約は、

`layout_dist` 変数 1 変数 2 `distance`

のように記述することができる。ここで、変数 1 と変数 2 は終端シンボルもしくは非終端シンボルの属性、 `distance` は図形間の距離を表す定数である。変数 1 と変数 2 の型としては integer、 point を用いることができる。変数 1 の値と `distance` ほど離れている座標を求めて、変数 2 の値に代入したあと変数 2 を属性として持つ図形を変った位置に描画する。例えば、図形の構成要素としてノード `N1` と `N2` があるとする。 `layout_dist N1.mid_x N2.mid_x 100` は、 `N1` と `N2` の構成要素が解釈された後、 `N1` の構成要素の属性 `mid_x` （中心の x 座標）と `N2` の構成要素の属性 `mid_x` との距離を 100 ドットにして `N2` の構成要素を描画する。

2.2.2 硬いレイアウト制約の扱い方

硬いレイアウト制約は、 CMG に基づいた制約の拡張として考えることができる。その理由は、硬いレイアウト制約は通常の制約のように生成規則が適用される特定の図形要素間に与えられる制約だからである。

すなわち、生成規則の定義、

$$
\begin{aligned}
T(\vec{x}) \ ::= \ & T_1(\vec{x_1}), \cdots, T_n(\vec{x_n}) \ where \\
& exists \ T'_1(\vec{x'_1}), \cdots, T'_m(\vec{x'_m}) \\
& where \ C(\vec{x_1}, \cdots, \vec{x_n}, \vec{x'_1}, \cdots, \vec{x'_m}) \ and \\
& HC(\vec{x_1}, \cdots, \vec{x_n}) \ and \\
& \vec{x} \ = \ F(\vec{x_1}, \cdots, \vec{x_n}, \vec{x'_1}, \cdots, \vec{x'_m})
\end{aligned}
$$

の中で硬いレイアウト制約（HC）は、通常の制約（C）のところに記述する。これは、硬いレイアウト制約が CMG に基づいた制約と同様の性質を持つものだからであり、通常の制約の延長として扱うことができる。

例えば、図 2.2（a）のように円（Node）とそれらを結ぶ直線（Edge）から構成されるグラフ構造があるとする。また、図 2.2の各ノードのとなりに書いてある数字は図形の「構成要素の種類. 構成要素の順番」だとする。図 2.2（b）のようにエッジでつながっているノード間の距離を 200 して各ノードを四角にレイアウトしたい場合、次のように生成規則を定義する。

```
1:graph() ::= N1:Node, N2:Node, N3:Node, N4:Node,
2:             E1:Edge, E2:Edge, E3:Edge
3:   where (
4:       E1.start == N1.mid  &&
5:       E1.end   == N2.mid  &&
6:       E2.start == N2.mid  &&
7:       E2.end   == N3.mid  &&
8:       E3.start == N3.mid  &&
9:       E3.end   == N4.mid  &&
```

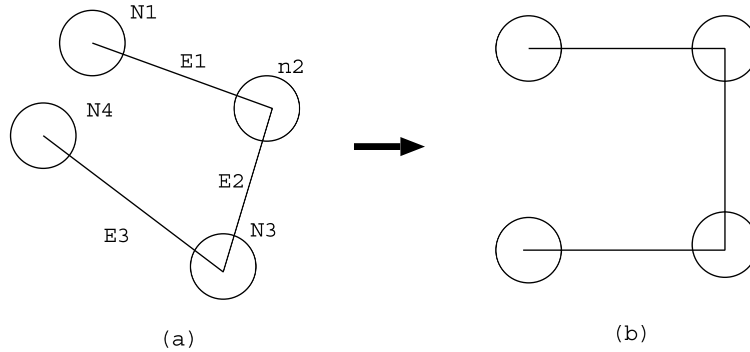

図 2.2: 硬いレイアウト制約の例

```
10:       layout_eq    N1.mid_y  N2.mid_y &&
11:       layout_dist  N1.mid_x  N2.mid_x  200 &&
12:       layout_eq    N2.mid_x  N3.mid_x &&
13:       layout_dist  N2.mid_y  N3.mid_y  200 &&
14:       layout_eq    N3.mid_y  N4.mid_y &&
15:       layout_dist  N3.mid_x  N4.mid_x  -200
16:   )
```

2.2.3 通常の制約と硬いレイアウト制約の違い

図形の解釈が成功するためには、文法に書かれた通常の制約を満す必要がある。また、通常の制約は図形を解釈することにより、図形間の関係をそのまま維持する制約である。それに対して、レイアウト制約は図形が解釈された後、もともとの図形要素間になかった位置関係を作り出す制約である。

通常の制約の中で eq 制約がある。 eq 制約は、図形を解釈するときに予め二つの図形の属性の値が等しい（equal）ことを意味する。この制約が一度成り立つと、一つの属性の値が変っても常にこの制約が成り立つように他の属性の値が書き換えられる。それに対して、 layout_eq は、図形を解釈するときに異なっていた二つの図形の属性の値を解釈が成功した後等しくする。

硬いレイアウト制約を導入した理由は、生成規則により生成される非終端シンボルの一部をローカルにレイアウトするためである。また、硬いレイアウト制約と軟

かいレイアウト制約を混ぜて用いることにより、空間パーサの応用が広がると考えられる。

2.2.4 空間パーサとレイアウト制約との関係

空間パーサとレイアウト制約との関係を明確にするため、「空間パーサとレイアウト機能を別々にしたシステム」について考察し、「空間パーサにレイアウト制約を追加したシステム（我々が提案したシステム）」と比較を行うことにより、「我々が提案したシステム」の特色を明らかにすることを試みる。

「空間パーサとレイアウト機能を別々にしたシステム」には、様々な作り方が可能であると思われるが、最も自然な作り方は、まず空間パーサにより、図形からノードやエッジに相当する非終端シンボルを認識し、次に、ユーザが定義したレイアウト規則により、認識したノードやエッジをレイアウトするシステムであろう。

この場合、空間パーサは、図形からノードやエッジに相当する非終端シンボルを認識した時点で、情報をレイアウトシステムに渡してしまうので図形のレイアウトは可能であるが、正確には与えられた図形がグラフ構造であるかどうかのパーシングは行っていない。通常、グラフ構造のパーシングを行うためには再帰的な生成規則により、文法を定義する必要がある。しかしながら、グラフ構造を再帰的に最後までパーシングすると一つの非終端シンボルしか存在しなくなる。この場合、パーシングしながらレイアウトすることが困難である。

一方、「我々が提案したシステム」では、軟らかいレイアウトの生成規則を用いて図形のパーシングとレイアウトを同時に行うことができる。

また、再帰的に定義された生成規則が図形に適用された場合、例としてあげたリスト構造のように、低い階層を持つ非終端シンボルから高い階層を持つ非終端シンボルを生成して行く。「我々が提案したシステム」では、空間パーサにレイアウト制約を追加することにより、まず軟らかいレイアウト制約や硬いレイアウト制約が与えられた図形を非終端シンボルとして認識することができる。これを低い階層を持つ非終端シンボルのレイアウトだと考える。また、その非終端シンボルにさらに軟らかいレイアウト制約や硬いレイアウト制約を与えることができる。すなわち、ノードやエッジとして扱おうとする図形に硬いレイアウト制約を与え、ローカルなレイアウトを行い、非終端シンボルとして認識する。次に、ノードやエッジの全体

に軟らかいレイアウトの生成規則を適用し、グラフ描画アルゴリズムに従ってバランスよくレイアウトすることが可能である。従って、「我々が提案したシステム」では、階層的にレイアウトすることが容易である。

しかしながら、「空間パーサとレイアウト機能を別々にしたシステム」では、階層的にレイアウトすることが困難である。あえて階層的にレイアウトするためには、まず低い階層を持つ非終端シンボルのパーシングが終ってからレイアウトを行い、次に、より高い階層を持つ非終端シンボルのパーシングが終ってからレイアウトを行うというように図形のパーシングとレイアウトを繰り返し行う必要があるが、これらを実装することは「空間パーサとレイアウト機能を別々にしたシステム」では難しい。

第 3 章

システム「Rainbow」

3.1 「Rainbow」の概要

「Rainbow」では、図 3.1のような図形エディタを備えている。図形エディタは、図形言語の文法を定義する場合と実際に図形言語を実行する場合に用いられる。

「Rainbow」で生成規則を定義するときに、ユーザは一つの非終端シンボルとしたい図形を図形エディタに描く。次に、その図形を選んで CMG 入力ウィンドウ（図 3.2）を開く。そうすると「Rainbow」は図形から構成要素とそれらの属性間に成り立っている制約を CMG 入力ウィンドウに書き出す。 CMG 入力ウィンドウは上から順番に非終端シンボルの名前、属性、制約、構成要素を書く欄になっている。ここにユーザは制約を修正し、また非終端シンボルの名前、属性を追加することにより、生成規則を定義していく（図 3.3）。

CMG 入力ウィンドウに書かれる制約としては、 `eq`（equal）、 `neq`（not equal）、 `gt`（greater than）、 `ge`（greater or equal）、 `lt`（less than）、 `le`（less or equal）、 `vp_close` がある。これらの制約を常に成り立たせるための機構を制約解消系と呼ぶ。制約解消系としては SkyBlue[23] を用いている。 CMG 入力ウィンドウの中で、制約の書き方は「制約名 変数 1 変数 2」である。ここで、変数 1 と変数 2 は定義している非終端シンボルの構成要素になる終端シンボルもしくは非終端シンボルの属性を示している。

`eq` 制約は、二つの変数 1 と変数 2 の値が等しいことを意味する。この制約が一度成り立つと一つの変数の値が変っても制約解消系によって常にこの制約が成り立つように他の変数の値が書き換えられる。 `neq` 制約は、二つの変数 1 と変数 2 の値

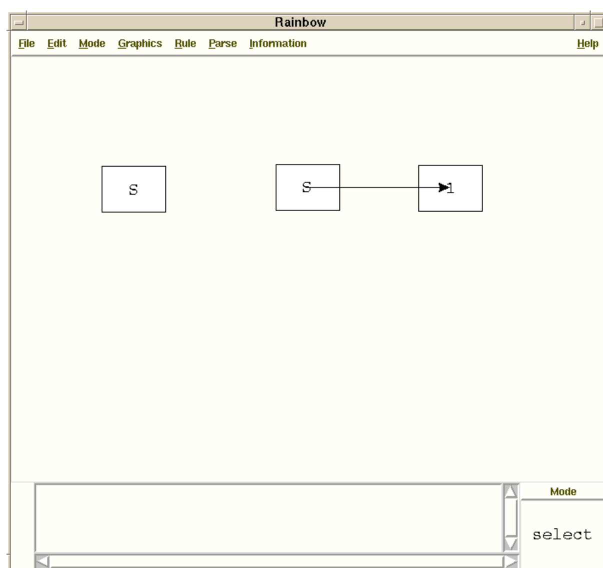

図 3.1: 「Rainbow」の図形エディタ

が等しくないことを表す。この制約は一度成り立っても一つの変数の値が変ると維持されない。 gt 制約は変数 1 の値が変数 2 の値より大きいこと、 ge 制約は変数 1 の値が変数 2 の値より大きいか等しいことを意味する。 lt 制約は変数 1 の値が変数 2 の値より小さいこと、 le 制約は変数 1 の値が変数 2 の値より小さいこか等しいことを表す。 vp_close 制約は、変数 1 の値と変数 2 の値がある程度近いことを意味する。この制約が一度成り立つと eq 制約が課せられ、一方の値が変化すると他方も同じ値に変化する。

属性の参照は、「構成要素の種類. 構成要素の順番. 属性名」の形で行う。構成要素の種類は構成要素になる終端シンボルもしくは非終端シンボルが normal （exist）の構成要素だったら 0 （1）になり[1]、構成要素の順番は構成要素の種類の中で何番目の構成要素かを表す（0 から始まる）。例えば、 normal の構成要素の 2 番目の構成要素の属性 mid （中心）を表す場合には「0.1.mid」のように記述する。

図 3.1の上左の図形はリスト構造の生成規則 1 を定義するため入力した図形で、上右の図形はリスト構造の生成規則 2 を定義するための図形である。上右の図形を選択して開かれた CMG 入力ウィンドウが図 3.2である。

[1]not_exist は 2、 all は 3 になる。

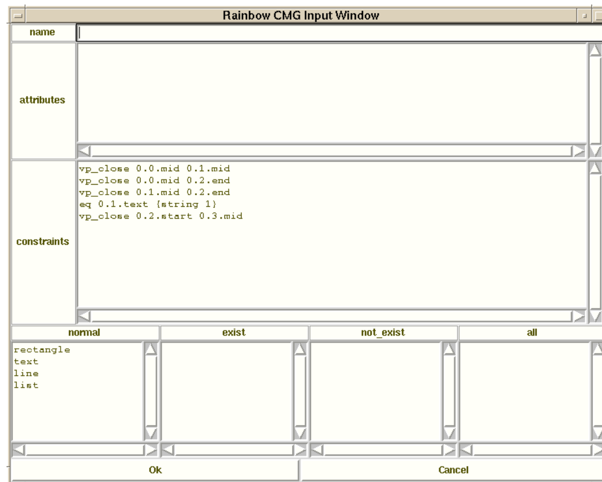

図 3.2: 「Rainbow」の CMG 入力ウィンドウ (1)

図 3.2の CMG 入力ウィンドウの構成要素の欄には、以下のように書かれる。

```
rectangle
text
line
list
```

また、制約の欄には以下のような制約が書かれる。

```
vp_close 0.0.mid 0.1.mid
vp_close 0.0.mid 0.2.end
vp_close 0.1.mid 0.2.end
eq 0.1.text {string 1}
vp_close 0.2.start 0.3.mid
```

ここで、1 行目は rectangle（0.0）の中心（mid）が text（0.1）の中心とほぼ一致していることを表す。2 行目は rectangle の中心が line（0.2）の終点とほぼ一致していることを表す制約である。3 行目は text の中心と line の終点（end）がほぼ一致していることを表す。4 行目は text の文字列（text）か 1 であるという制約である。5 行目は line の始点（start）か list（0.3）の中心がほぼ一致していることを表す。

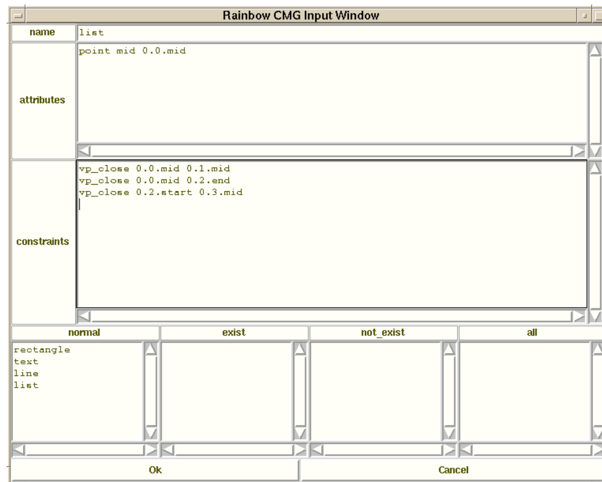

図 3.3: 「Rainbow」の CMG 入力ウィンドウ (2)

ユーザは、この CMG 入力ウィンドウの非終端シンボルの名前の欄に `list` と書き、非終端シンボルの属性の欄には `list` の属性 `mid` を `rectangle` の中心にするように書く。

```
point mid 0.0.mid
```

また、制約の欄には 1 行目、 2 行目、 5 行目の制約を選ぶ（図 3.3）。これで、リスト構造の生成規則 2 が定義される。

我々は、軟かいレイアウトの生成規則を定義するときにも、通常の生成規則を定義する場合と同じように、図形を用いて行う。まず、ユーザは解釈したい図形を図形エディタに描いてその図形またはその一部を選択し（図 3.4の上の図形）、「軟かいレイアウト CMG 入力ウィンドウ（図 3.5）」を開く。このウィンドウは上から各軟かいレイアウト制約の定数を設定するメニュー（Layout Constant Input）、非終端シンボルの名前、軟らかいレイアウト制約の名前 SC、再帰的に生成される非終端シンボルの名前 GS、 `node` の構成要素の名前、 `edge` の構成要素の名前を書く欄になっている。「Rainbow」では、図形を選択すると、 `node` と `edge` の構成要素の名前の欄には、四角、円、直線などの終端シンボルを除いて非終端シンボルの名前がシステムにより自動的に以下のように書き出される（図 3.5）。

```
lNode
```

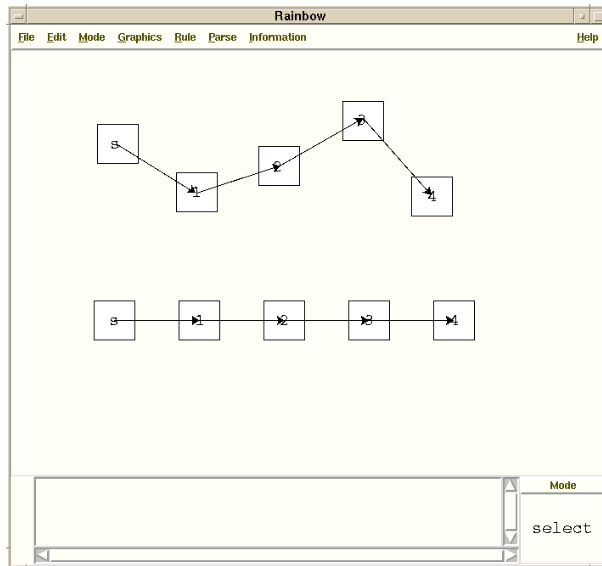

図 3.4: レイアウト前後のリスト構造

`lEdge`

`list`

図 3.5は、リスト構造（`list`）の構成要素ノード（`lNode`）とエッジ（`lEdge`）を認識する生成規則があらかじめ存在した場合に、開いた軟かいレイアウト CMG 入力ウィンドウである。

次に、ユーザは `node` や `edge` の構成要素にしたい非終端シンボルの名前を選び、非終端シンボルの名前、SC、GS を定義する。非終端シンボルの名前として `listModel`、SC として `listStructure`、GS として `list` を定義する。また、指定するレイアウトの定数を設定する（図 3.6）。レイアウトの定数を設定しなかった場合には、「Rainbow」で用意した定数の値が使われる。これで、軟かいレイアウトの生成規則が定義される。

図形言語の生成規則の定義が終わったら実際に図形を図形エディタに入力し、パーシングすることが可能になる。一般に「Rainbow」では、図形を解釈するモードとして自動モードと要求モードの二種類を用意している。新たな図形の入力があるたびにパーシングを行うのが自動モード、また、ユーザからの要求があったときのみにパーシングを行うのが要求モードである。

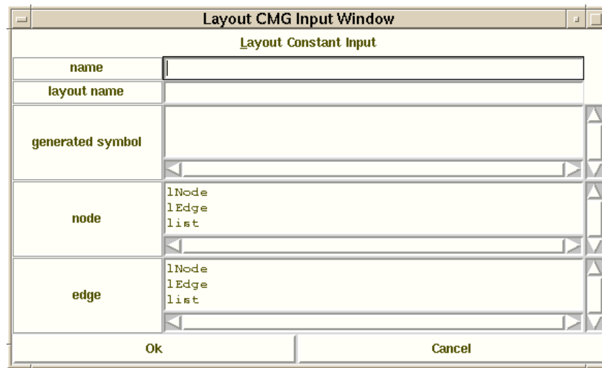

図 3.5: リストの軟かいレイアウト CMG 入力ウィンドウ (1)

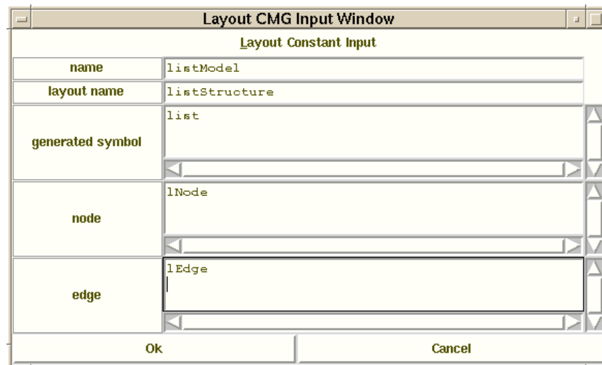

図 3.6: リストの軟かいレイアウト CMG 入力ウィンドウ (2)

定義した生成規則を用いて、図 3.4の上の図形を解釈すると下の図形のようにレイアウトされる。

3.2 「Rainbow」の構造

図 3.7は、「Rainbow」の構成要素を表している。「Rainbow」への入力は、通常の生成規則、レイアウトの生成規則、解釈したい図形で、「Rainbow」からの出力は、入力の図形をレイアウトした図形である。

図形言語の通常の生成規則と軟らかいレイアウトの生成規則は、「Rainbow」により内部表現が作られ、生成規則の集合に保存される。図 3.8は、生成規則の内部表現を表している。ただし、 P は連想配列で、 *p-id* は一つの生成規則に付けられ

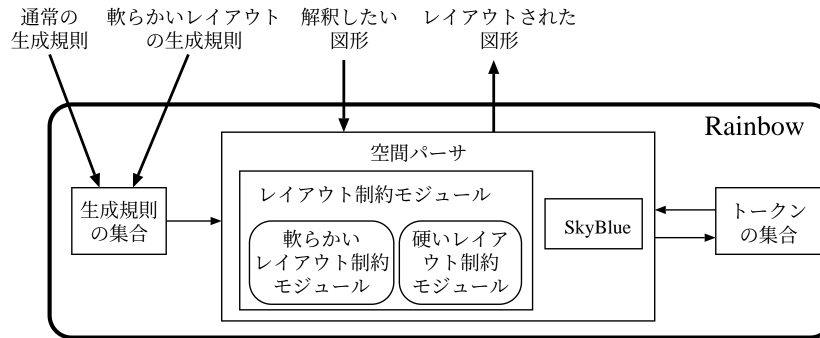

図 3.7: 「Rainbow」の構成図

名前	値
P(*p-id*.name)	非終端シンボルの名前
P(*p-id*.attrs)	非終端シンボルの属性のリスト
P(*p-id*.constraints)	構成要素の属性間の制約のリスト
P(*p-id*.negative_constraints)	ネガティブ制約のリスト
P(*p-id*.normal)	非終端シンボルの normal の構成要素のリスト
P(*p-id*.exist)	非終端シンボルの exist の構成要素のリスト
P(*p-id*.not_exist)	非終端シンボルの not_exist の構成要素のリスト
P(*p-id*.all)	非終端シンボルの all の構成要素のリスト

図 3.8: 生成規則の内部表現

た id である。

また、解釈したい図形を「Rainbow」に与えると、空間パーサにより図形を構成する全ての終端シンボルのトークンはトークンの集合に保存される。さらに、トークンの内部属性間に存在する制約は制約解消系 SkyBlue[23] に追加される。ここで、トークンの内部属性間に存在する制約として、例えば、四角を表すトークンの中心の x 座標は、四角の左上の x 座標と右下の x 座標を足して 2 で割るなどの制約がある。各トークンはトークンの属性を内部データとして持つ。トークンの属性は、属性の型と値を用いて新たな SkyBlue の変数を作成し、その SkyBlue の変数を値として持つ。トークンの属性値を SkyBlue の変数とするので、レイアウト制約を実現するためのレイアウト制約モジュールによりトークンの属性値が変ったときでも制約解消を SkyBlue に行わせることが可能である。

3.3 「Rainbow」の解釈アルゴリズム

CMG で記述された生成規則の集合として文法を記述し、解釈の対象となる図形を与えると通常は以下のように解釈が進む [3]。まず、ユーザが記述した生成規則と軟かいレイアウトの生成規則は、生成規則の集合 SCC に保存される。また、解釈の対象となる図形を構成する全ての終端シンボルのトークンは、トークンの集合 ParseForest に格納される。さらに、それらのトークンの内部属性間に存在する制約は制約解消系 SkyBlue に追加される。

実際の図形の解釈は、以下の [1] を行い、 [1] が終了したあとで [2] を行うことにより実行する。

[1] ParseForest が変更されなくなるまで、 SCC の各通常の生成規則に対して、構成要素の候補になれるトークンの組合わせリストを ParseForest から求めることを繰り返す。次に、各トークンの組合わせに対して、生成規則に記述された制約を満し、さらに硬いレイアウト制約が存在する場合にその解が存在するかを繰り返し検査を行う。もし、生成規則に記述された制約および硬いレイアウト制約を満す場合には、その生成規則が適用され、硬いレイアウト制約モジュールが実行される。そして、新たな非終端シンボルのトークンを ParseForest に挿入し、生成に用いられたトークンを ParseForest から削除する（トークンに「削除マーク」を付けるだけで実際には ParseForest から削除されない）。また、全ての制約を SkyBlue に追加する。

[2] SCC の各軟かいレイアウトの生成規則に対して、 GS （再帰的に生成される非終端シンボルの名前）が ParseForest に存在するかを検査し、 GS の構成要素が GS を持たなくなるまで繰り返し検査し、 `node` や `edge` の構成要素のマルチセットを求める。

求められた `node` や `edge` の構成要素のマルチセットに SC が与えられ、レイアウトが行われる。さらに、 S （新たに生成された非終端シンボルの名前）のトークンは ParseForest に挿入される。

図 3.9に、 [1] と [2] を分かりやすくまとめた。

```
repeat
  foreach 通常の生成規則
    トークンの組み合わせリストを求める
    foreach トークンの組み合わせ
      if 通常の制約と硬いレイアウト制約を満たすか then
        硬いレイアウト制約のモジュールの実行
        新たに生成されたトークンをトークンの集合に追加
        生成に用いられたトークンをトークンの集合から削除
        制約を SkyBlue に追加
      end if
    end foreach
  end foreach
until トークンの集合が変更しない

foreach 軟らかいレイアウトの生成規則
  if GS がトークンの集合に存在するか then
    repeat
      node や edge の構成要素のマルチセットを求める
    until GS の構成要素が GS を持ったなくなる
    軟らかいレイアウト制約を与える
    新たに生成されたトークンをトークンの集合に追加
  end if
end foreach
```

図 3.9: 解析アルゴリズム

3.4 レイアウト制約モジュール

3.4.1 軟らかいレイアウト制約モジュール

軟かいレイアウト制約モジュールの実行により軟かいレイアウト制約の処理が行われる。スプリングモデル制約モジュールでは、 GS の構成要素が GS を持ったなくなるまで繰り返し検査し、 node や edge の構成要素のマルチセットを求め、図形要素間のグラフ構造を得る。グラフ構造を用いて隣接するノード間に働く力 f_s、隣接しないノード間に働く力 f_r が求められる。

マグネティックスプリングモデル制約モジュールでは、スプリングモデルに必要なグラフ構造に加えて、それぞれのエッジの種類を決めることとそのエッジが磁場に対して指す方向が必要である。これらを用いて f_s、 f_r、エッジが受ける回転力 f_m が求められる。

リスト構造制約モジュールでは、 GS の構成要素が GS を持ったなくなるまで繰り返し検査し、 node や edge の構成要素のマルチセットを求め、図形要素間のグラフ構造を得る。ヘッドのノード（テキストとして s を持つノード）を決められた位置に配置し、あらかじめ与えられた順序に従いヘッドのノードの平行線上にノードを左から右に並べる。ノードの y 座標はヘッドのノードの y 座標と同じであり、ノードの x 座標は連続するノードの間を一定の間隔に開けるようにする。

tree 制約モジュールは、 GS の構成要素が GS を持ったなくなるまで繰り返し検査し、 node や edge の構成要素のマルチセットを得て、根ノード、ノード間の親子関係や兄弟関係などの情報を求める。次に、これらの情報を用い、 Walker の一般木描画アルゴリズム [22] によってノードの最終座標を計算を行う。このアルゴリズムでは、木の根となる親ノードは描画の最上の位置に配置され、親ノードの各子ノードは下方の階層に対応した平行線上に、あらかじめ与えられた順序（後順）に従い左から右に並べられる。

3.4.2 硬いレイアウト制約モジュール

layout_eq 制約モジュールでは、このモジュールの引数になる変数 2 の値として変数 1 の値を代入し、変数 1 と変数 2 に SkyBlue の eq 制約を与える。 eq 制約は、一度成り立つと一つの変数が変っても SkyBlue によって常にこの制約が成り立つ

ように他の変数の値を変った値に等しくする。

layout_dist 制約モジュールでは、このモジュールの引数になる変数 1 の値に変数 3 の値を足して変数 2 の値として代入し、 SkyBlue の plus 制約を与える。 plus 制約は、一度成り立つと一つの変数が変っても常にこの制約が成り立つように他の変数の値を変えて一定の距離を維持させる。

3.4.3 「Rainbow」と恵比寿の比較

恵比寿の図形エディタは、図形言語の文法を定義する定義ウィンドウと実際に図形の解釈を行う実行ウィンドウに分れていたが、「Rainbow」では一つのウィンドウに統合した。なぜなら、ユーザは文法を定義しながら図形を描いて文法の正しさを検査することの繰り返しによって文法を定義していくので、文法の定義モードと図形の実行モードを一緒にする方が便利である。そうすれば、一つの非終端シンボルとしたい図形の中に他の生成規則によって定義された非終端シンボルを認識することができる。また、複雑な図形を表示するのに画面の空間を有効に使える。

「Rainbow」では、恵比寿が扱う制約の他にレイアウト制約を実現した。特に、軟かいレイアウト制約の導入により、図形の全体を解釈しながらインタラクティブにバランスよくレイアウトすることが可能になっている。

第 4 章

「Rainbow」の適用例

4.1　スプリングモデル制約

4.1.1　スプリングモデル

スプリングモデル [20] は無向グラフを描画するためのアルゴリズムの中で最も基本となるアルゴリズムである。本アルゴリズムは、エッジの長さを一定にすることやノード間の対称性を顕示することを目指して無向グラフを描画する場合に多く応用される。

スプリングモデルとは、ノードを鉄のリングに、エッジを力学系を形成するバネに置き換え、リングの安定状態を見つけることで適切なレイアウトを求めるものである（図 4.1）。

このモデルは 2 種類のバネが使われる。すなわち隣接するノード間をつなぐバネと隣接しないノード間に斥力だけを与えるバネである。隣接するノード間に働く力 f_s は

$$f_s \quad = \quad c_1 \log(d/c_2)$$

により与えられる。ここで d はノード間の距離、 c_1 と c_2 は定数とする。 $d > c_2$ のとき f_s は引力、 $d < c_2$ のとき斥力、 $d = c_2$ のときノード間に力は働かない。また、隣接しないノード間に働く力 f_r は

$$f_r \quad = \quad c_3/d^2$$

により与えられる。ここで c_3 は定数とする。このような力 f_s と f_r をできるだけ緩

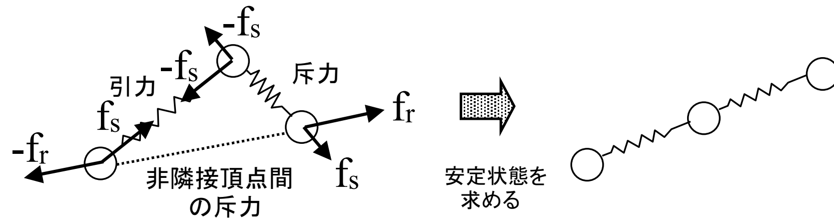

図 4.1: スプリングモデルによるグラフのレイアウト

和するように各ノードを (c_4 × そのノードに働く力) ずつ移動させることにより、すべての隣接するノード間の距離を c_2 の近傍に収束させる。 c_4 は定数とする。

4.1.2 スプリングモデル制約の例

スプリングモデル制約の例として、データベース分野で実世界のデータ構造を記述するのに用いられる E-R ダイアグラム [24] を考える。 E-R ダイアグラムの構成要素を次のように定義する。 1) 四角の中に実体名が書いてある図形を実体ノード（`entityNode`）とする。 2) 円の中に属性名が書いてある図形を属性ノード（`attributeNode`）とする。 3) 実体ノード間の関連を表す直線を実体エッジ（`entityEdge`）とする。その直線の中心には関連型が書いてある。 4) 実体ノードと属性ノード間の関係を表す直線を属性エッジ（`attributeEdge`）とする。これらの構成要素を解釈する生成規則を「Rainbow」を用いて定義する。例えば、実体ノードを解釈する生成規則の場合、ユーザは「Rainbow」の図形エディタに四角を描いてその中にテキストを書いて CMG 入力ウィンドウを開く。 CMG 入力ウィンドウの非終端シンボルの名前の欄には、 `entityNode` を書く。属性の欄には、実体ノードの属性 `mid` （中心）を四角の中心にするように書く。制約の欄には、四角の中心とテキストの中心を等しくする制約を選び、非終端シンボル `entityNode` の定義を完了する。さらに、非終端シンボル `ERGraph` を生成するように、 E-R ダイアグラムを再帰的に定義する生成規則を定義する。

次に、解釈したい図 4.2の図形を選んで軟かいレイアウト CMG 入力ウィンドウを開くと、システムは、 `node` と `edge` の構成要素の名前の欄に、四角、円、直線などの終端シンボルを除いて非終端シンボルの名前を自動的に書き出す。 `node` と `edge` の構成要素の名前の欄には

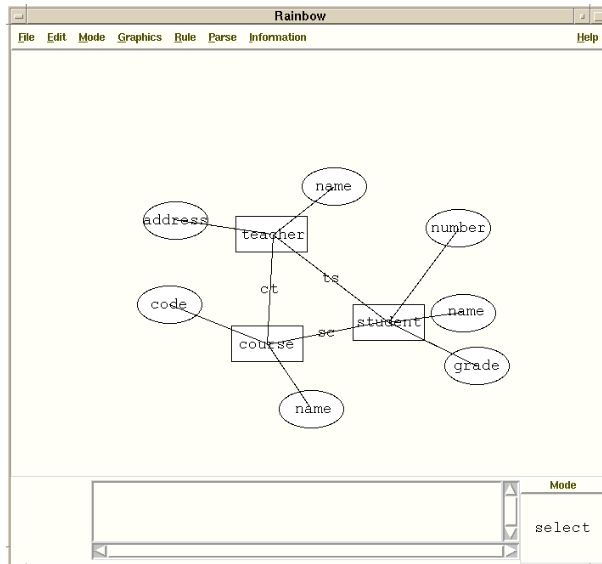

図 4.2: レイアウト前の E-R ダイアグラム

entityNode

attributeNode

entityEdge

attributeEdge

のように記述される。そうすると、ユーザは node の構成要素の名前の欄には

entityNode

attributeNode

edge の構成要素の名前の欄には

entityEdge

attributeEdge

を選ぶ。

次に、ユーザは「Layout Constant Input」メニューを選択してスプリングモデルの定数を決める。非終端シンボルの名前の欄には ERModel、軟らかいレイアウト制約の名前の欄には spring、再帰的に生成される非終端シンボルの名前の欄には

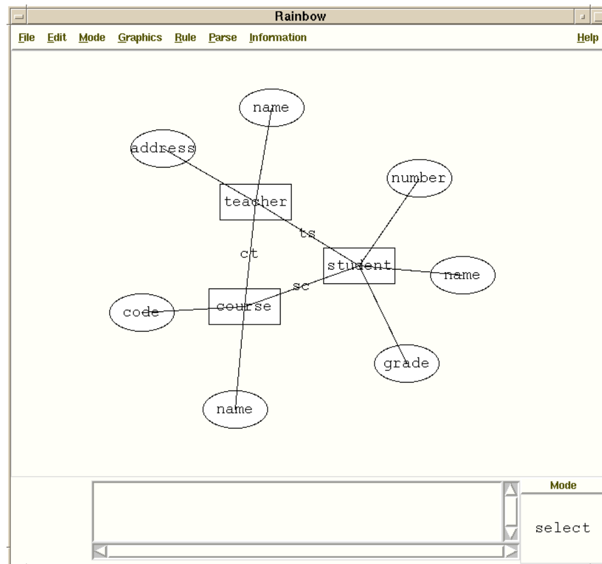

図 4.3: レイアウト後の E-R ダイアグラム

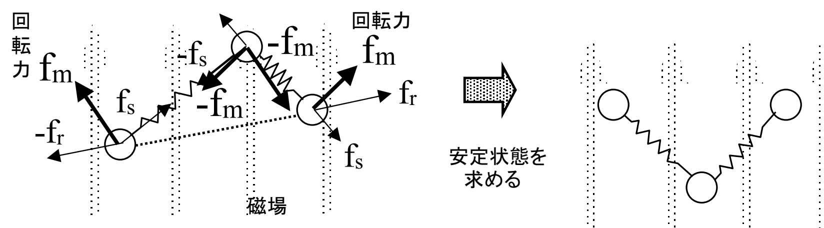

図 4.4: マグネティックスプリングモデルによるグラフのレイアウト

ERGraph を書いて軟かいレイアウトの生成規則を定義する。これらの生成規則を用いて、図 4.2の図形を解釈すると図 4.3のようにレイアウトされる。

4.2 マグネティックスプリングモデル制約

4.2.1 マグネティックスプリングモデル

マグネティックスプリングモデル [21] は、スプリングモデルに磁場の概念を入れたものである（図 4.4）。エッジを磁針と見なし、グラフの置かれた磁場から回転力を受ける。マグネティックスプリングモデルの磁場には、有向エッジに働くものと無向エッジに働くものの二種類がある。有向エッジの場合はエッジの終点が磁場

の北を向くように回転力を受ける。無向エッジの場合は磁場の向きは関係なくノードは南北の向きに近い方を向くように回転力が働く。回転力は次のように定義される。

$$f_m = c_5 b d^\alpha \mid t \mid^\beta$$

ここで、 b は基準点（エッジの中心）における磁場の強さ、 d は現在のエッジの長さである。 t はエッジの基準点における磁場の北からの終点のずれの角度である。すなわち、有向エッジの場合は $0 < t \leq \pi$、無向エッジの場合は $0 < t \leq \pi/2$ となる。また、 α は辺の長さの回転力への影響を制御する定数、 β は t の回転力への影響を制御する定数である。また、隣接するノード間に働く力と隣接しないノード間に働く力は、 f_s と f_r を用いる。

マグネティックスプリングモデルは、複数種類のエッジを持つ有向グラフのレイアウトにおいてそれぞれのエッジを一定の方向に向かせることや、有向エッジと無向エッジが混在したグラフのレイアウトにおいて有向エッジだけを一定の方向に向かせ、また、無向エッジも特定の方向にそろえることの応用に用いられる。

4.2.2 マグネティックスプリングモデル制約の例

マグネティックスプリングモデル制約の例として、オブジェクト指向に基づくソフトウェア設計に用いられるオブジェクト図 [11] [25] の自動レイアウトについて考える。オブジェクト図の構成要素として、クラス（class）、関連（association）、汎化（generalization）、および、集約（aggregation）が存在する。ここで、クラスをノード、 3 種類の関係をエッジとして見なすことができ、 3 種類のエッジをそれぞれ違う方向に向けるようにすれば、オブジェクト図の自動レイアウトが可能である [26]。

それぞれのエッジへどの種類の磁場を与えるかは、オブジェクト図におけるそれぞれのエッジの特性や意味的な要素から以下のようにする。関連は、オブジェクト同士の対等な参照あるいは利用関係を表すもので、方向のない関係である。従って関連エッジは平行磁場とし、横向きに磁場を与える。汎化は、クラス間の概念的な包含関係、すなわちスーパークラスとサブクラス間の継承関係を表すものなので、上から下の有向関係である。従って汎化エッジは下向の磁場を与える。集約は、一

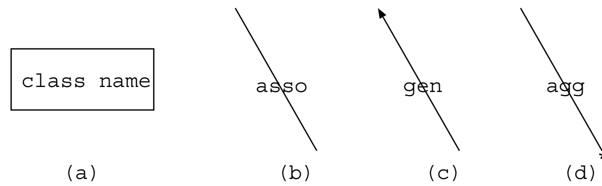

図 4.5: オブジェクト図の構成要素

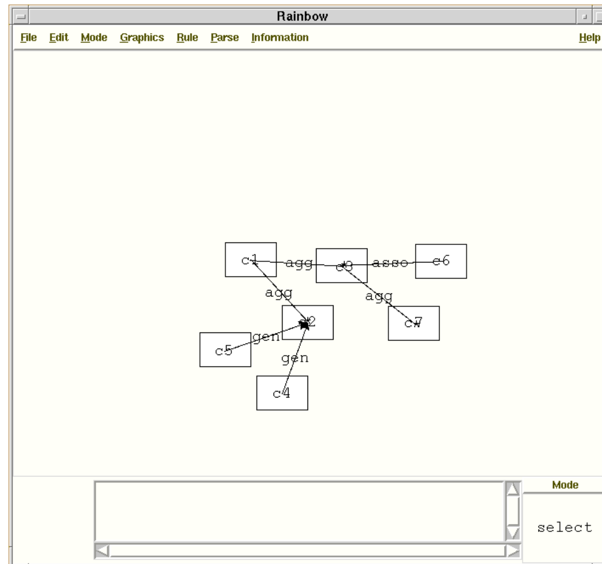

図 4.6: レイアウト前のオブジェクト図

方のオブジェクトが他方の部分となるような構造的な包含関係を表すものである。集約エッジは汎化エッジと区別するために斜め右下 45 度の磁場を与える。

　我々は、オブジェクト図の構成要素を区別するため図 4.5のように定義する。（a）は四角の中にクラス名が書いてある図形をノードとして定義する。（b）は直線の中心に「asso」が書いてある図形を関連エッジとして定義する。（c）は直線の始点に矢印があって直線の中心に「gen」が書いてある図形を汎化エッジとして定義する。（d）は直線の終点に「＊」があって直線の中心に「agg」が書いてある図形を集約エッジとして定義する。まず、これらのオブジェクト図の構成要素を解釈する生成規則を定義する必要がある。定義方法は、図 4.5のように図形を図形エディタに描いて開いた CMG 入力ウィンドウで行う。例えば、集約エッジの場合は、非

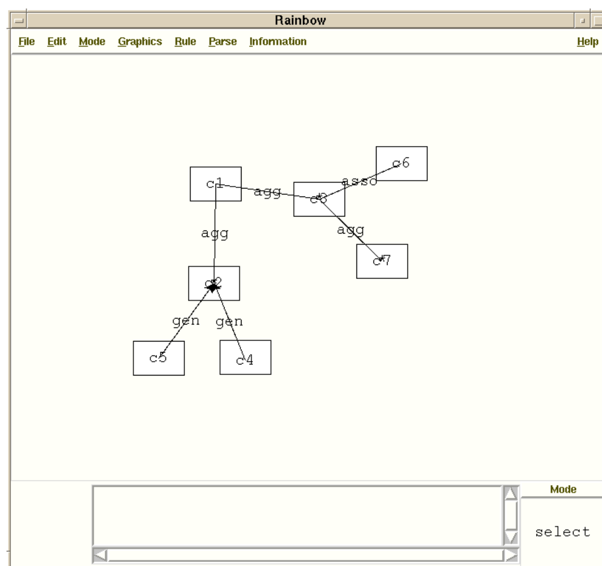

図 4.7: レイアウト後のオブジェクト図

終端シンボルの名前として CMG 入力ウィンドウの名前の欄に `aggregation` と書き、属性 `start`、 `end` は直線の始点、終点にするように書く。制約の欄には、直線の中心とテキストの中心、直線の終点と「∗」の中心を等しくする制約を選び、非終端シンボル `aggregation` の定義を行う。さらに、非終端シンボル `ObjectGraph` を生成するように、オブジェクト図を再帰的に定義する生成規則を定義する。

また、解釈したい図 4.6の図形を選んで軟かいレイアウト CMG 入力ウィンドウを開く。ユーザは自動的に書き出された非終端シンボルの名前から `node` と `edge` の構成要素を選び、マグネティックスプリングモデルの定数を決める。名前として `ObjectModel`、軟らかいレイアウト制約の名前の欄には `magnetic`、再帰的に生成される非終端シンボルの名前の欄には `ObjectGraph` を書く。次に、それぞれの `edge` の構成要素にエッジの種類と磁場を与える。これらの与え方は、 `edge` の構成要素の名前の欄に選ばれた構成要素の名前の最初にエッジの種類と磁場の角度をユーザが付け加える。この例では次のように記述する。

```
{undirect(0) association}
{direct(90) generalization}
{direct(45) aggregation}
```

これらの生成規則を用いて図 4.6の図形を解釈すると図 4.7のような結果が得られる。

4.3 木構造制約

4.3.1 木構造

我々は、木の描画法として Walker の一般木描画アルゴリズム [22] を用いる。その理由は、木に関する描画法の研究として行われてきたいろいろなアルゴリズムの中で、現在最も優れていると考えられるからである [1]。

Walker の一般木描画アルゴリズムでは、木のルートとなる親ノードは描画の最上の位置に配置され、親ノードの各子ノードは下方の階層に対応した平行線上に、あらかじめ与えられた順序に従い左から右に並べられる。ノードの `y` 座標は連続する階層の間を一定の間隔を開けるから容易に求められる。従って、ノードの `x` 座標を決定することが重要である。木構造のレイアウト規則モジュールでは、 Walker の一般木描画アルゴリズムを実現するための必要な情報を連想配列 `W` に格納している。連想配列 `W` のそれぞれの要素は図 4.8のようになっている。本節では変数名など実際の文字列はタイプライタ体で，値が変るものはイタリックで表記する．ここで、 *t-id* はトークンにユニークにつけられた id である。

ノードの最終的な `x` 座標を求めるために、まず、後方順探索により、各ノードの仮 `x` 座標値と修正座標値を求める。これらの座標値の求め方について述べる。 1) ノード `B` が葉であり、左側に兄弟ノードを持たない場合、 `prelim(B)` と `modifier(B)` は 0 である。 2) ノード `B` が葉であり、左側に兄弟ノード `A` を持つ場合、 `prelim(B)` は、

`prelim(A) + A の大きさの半分 + B の大きさの半分 + 兄弟分離`

であり、 `modifier(B)` は 0 である。 3) ノード `B` が葉ではなく、左側に兄弟ノードを持たない場合、 `prelim(B)` は、

`(prelim(B の最も左側の子ノード) + prelim(B の最も右側の子ノード)) / 2`

であり、 `modifier(B)` は 0 である。 4) ノード `B` が葉ではなく、左側に兄弟ノードを持つ場合、 `prelim(B)` は 2) と同じ計算式で、 `modifier(B)` は、

名前	値
W(root)	ルートノード
W(node)	node の構成要素のリスト
W(edge)	edge の構成要素のリスト
W(height)	木の高さ
W(yDistance)	階層間の一定の間隔
W(siblingDistance)	兄弟ノード間の最小距離（兄弟分離）
W(subtreeDistance)	部分木間の最小距離（部分木分離）
W(t-id.parent)	親ノード
W(t-id.child)	子ノードのリスト
W(t-id.leftChild)	最も左の子ノード
W(t-id.rightChild)	最も右の子ノード
W(t-id.leftSibling)	左側の兄弟ノード
W(t-id.rightSibling)	右側の兄弟ノード
W(t-id.leftNeihbor)	左側の部分木で同じ深さの最も右のノード
W(t-id.depth)	ノードの深さ
W(t-id.prelim)	仮 x 座標値
W(t-id.modifier)	修正座標値
W(t-id.final)	最終 x 座標値

図 4.8: 木構造のレイアウト規則モジュールの内部表現

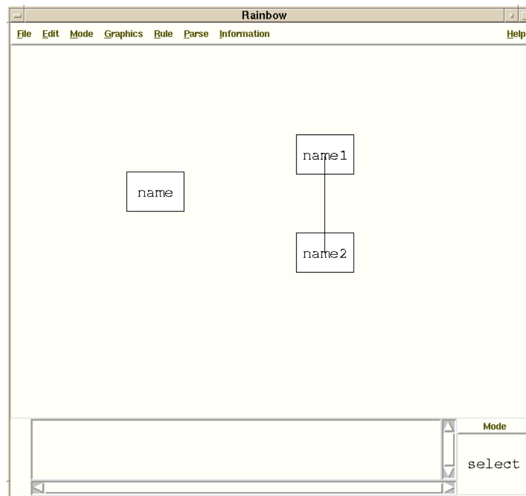

図 4.9: 「Rainbow」の図形エディタ

```
prelim(B) - (prelim(Bの最も左側の子ノード) +
prelim(Bの最も右側の子ノード)) / 2
```

である。仮 x 座標値と修正座標値を求めながら隣接する部分木間の分離を行う。部分木間の分離は、隣接するノードの根を兄弟分離の値だけ引き離し、一つ下がった階層で部分木分離が達成されるまで引き離す。この過程を短い方の部分木の最下層に着くまで順次繰り返す。

次に、先行順探索により、各ノードの仮 x 座標値にそのノードの祖先の修正座標値をすべて足し会わせることにより、各ノードの最終 x 座標値を計算する。最終 y 座標値は、各ノードの深さに階層間の一定の間隔をかけて求める。最終 x と y 座標値に従って図形を再描画する。

4.3.2 木構造制約の例

会社などの仕組みを表すのに用いられる組織図を解釈しながらレイアウトすることを考える。組織図は、部署を表すノードと部署間の包含関係を表すエッジで木構造に構成される。

「Rainbow」を用いて、組織図を解釈しながらインタラクティブに木構造にレイアウトするためには、組織図の構成要素を解釈する通常の生成規則の他に木構造のレイアウトの生成規則を定義する必要がある。「Rainbow」では、図形を用いて生

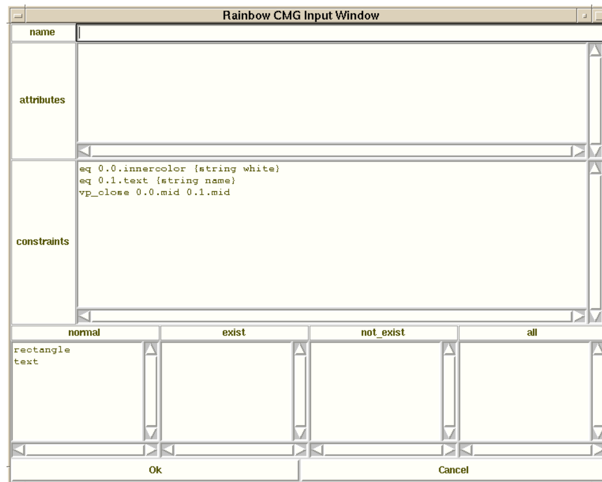

図 4.10: ノードの CMG 入力ウィンドウ (1)

成規則の定義を行っている。まず、ユーザは非終端シンボルノード（`tNode`）を生成する生成規則を定義するため、図形エディタに四角（`rectangle`）を描き、その中心に適当なテキスト（`text`）を書く（図 4.9の左の図形）。これらをまとめて選択し、図形エディタの「Rule」メニューから「Make New Production Rule」をクリックすると「CMG 入力ウィンドウ（図 4.10）」が開く。そのとき「Rainbow」によって、図形から構成要素とそれらの属性間に成り立っている制約は CMG 入力ウィンドウに書き出される。`normal` の構成要素の欄には、`rectangle` と `text` が書かれる。また、制約の欄（constraints）には以下のような制約が書かれる。

```
eq 0.0.innercolor {string white}
eq 0.1.text {string name}
vp_close 0.0.mid 0.1.mid
```

ここで、1 行目は `rectangle`（0.0）の塗り潰す色 `innercolor` が白であるという制約で、2 行目は `text`（0.1）の文字列か name であるという制約である。3 行目は、`rectangle` の中心（`mid`）と `text` の中心がほぼ一致していることを表す制約である。

我々は、この CMG 入力ウィンドウの非終端シンボルの名前の欄に `tNode` と書き、非終端シンボルの属性の欄には `tNode` の属性 `mid` を `rectangle` の中心にするよう

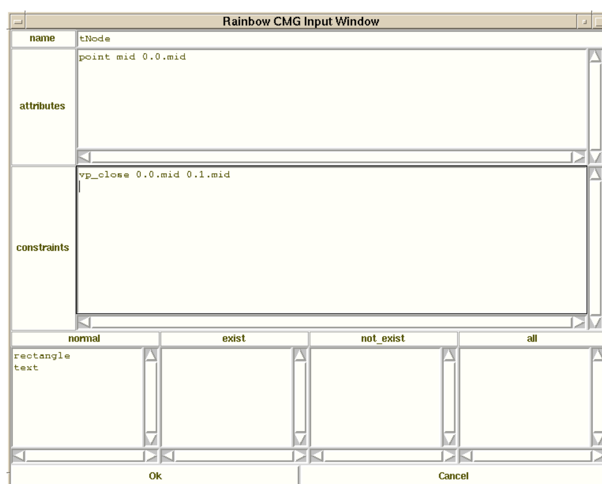

図 4.11: ノードの CMG 入力ウィンドウ (2)

に書く。また、制約の欄には、 `rectangle` の中心と `text` の中心を等しくする 3 行目の制約だけを選ぶ（図 4.11）。「OK」ボタンをクリックすることにより、 `tNode` の定義を完了する。同様にして非終端シンボルエッジ（`tEdge`）の生成規則を定義する。図 4.9の右の図形のように二つのノードとそのノード間を結ぶ直線（`line`）を描いて選択して開かれた CMG 入力ウィンドウに書かれた情報を修正した CMG 入力ウィンドウが図 4.12である。

さらに，非終端シンボル `organizationGraph` を生成するするように，組織図を再帰的に定義する生成規則を記述する。図 4.9の左の図形を用いて `tNode` を `organizationGraph` として認識する生成規則，図 4.9の右の図形を用いて二つの `organizationGraph` とそれらの間を結ぶ `tEdge` を `organizationGraph` として認識する生成規則を記述する。

次に、ユーザはレイアウトの生成規則を定義する。ユーザは図 4.13のような図形を図形エディタに描き、一部または全体を選択して図形エディタの「Rule」メニューから「Make New Layout Production Rule」をクリックする。そうすると、「軟かいレイアウト CMG 入力ウィンドウ（図 4.14）」が開く。「Rainbow」は、図 4.13の図形から四角、円、直線などの終端シンボルを除いて非終端シンボルの名前，すなわち `tNode`， `tEdge`， `organizationGraph`，を自動的に `node` と `edge` の構成要素の欄に書き出すので，ユーザは，それぞれの構成要素にしたい非終端シンボルの名

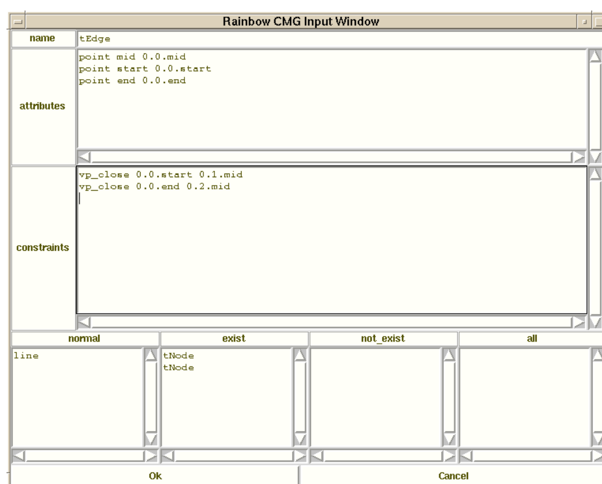

図 4.12: エッジの CMG 入力ウィンドウ

前を選ぶ。次に、ユーザは「Layout Constant Input」メニューを選択して階層間の一定の間隔、兄弟ノード間の最初距離などの定数を決める。非終端シンボルの名前の欄には `treeDiagrams`、軟らかいレイアウト制約の名前 SC の欄には `tree`, 再帰的に生成される非終端シンボルの名前 GS の欄には `organizationGraph` を書き、「OK」ボタンをクリックすることにより、木構造のレイアウトの生成規則の定義を完了する（図 4.15）。

解釈したい図形を実際に実行するには、レイアウトの生成規則を定義するときに用いられた図形またはその一部を選択し、図形エディタの「Parse」メニューから「Parse Selected Items」をクリックする。それにより、定義された生成規則が適用されると `tNode`， `tEdge`， `organizationGraph` が生成される。レイアウトの生成規則が適用されると、再帰的に生成される非終端シンボル GS の構成要素が GS を持ったなくなるまで繰り返し検査し、 `node` や `edge` の構成要素のマルチセットを求められ、図形を木構造にレイアウトする。図 4.13の図形を解釈すると、図 4.16のようなレイアウト結果が得られる。

4.3.3　木構造制約と硬いレイアウト制約を混ぜた例

木構造制約と硬いレイアウト制約を混ぜた例として、親族の関係を表すのに用いられる家系図を挙げる。家系図とは、人を表すノード（`node`）、夫婦関係を表す

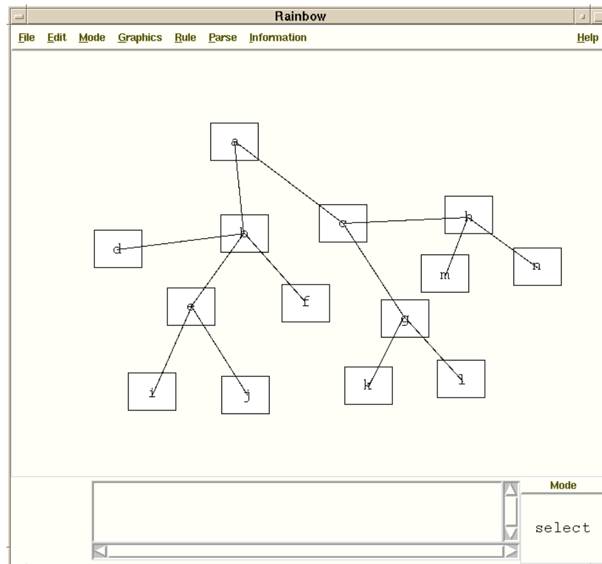

図 4.13: レイアウト前の組織図

親エッジ（pEdge）、親エッジとそのエッジに結ばれている二つのノードを表す親ノード（pNode）、親ノードとノードに親子関係を表す子エッジ（cEdge）で構成されている。これらの構成要素を解釈する生成規則を「Rainbow」を用いて定義する。そのとき、夫婦関係を分りやすく表すために、親ノードの構成要素の二つのノードを同じ平行線上に配置し、ノード間の距離を一定にすることが望まれる。そこで、我々はある図形を親ノードとして認識する生成規則の制約に `layout_eq` 制約と `layout_dist` 制約を用いる。さらに、非終端シンボル `kinshipGraph` を生成するように、家系図を再帰的に定義する生成規則を定義する。

また、子エッジと結ばれている親ノードとノードとの親子関係を分りやすくするため、木構造に表すことが望まれる。そこで、我々はレイアウトの生成規則の軟かいレイアウトの種類として `tree` を用いる。非終端シンボルの名前の欄には `kinshipDiagrams`、再帰的に生成される非終端シンボルの名前の欄には `kinshipGraph`、`node` の構成要素の名前の欄には `node` と `pNode`、 `edge` の構成要素の名前の欄には `cEdge` を用いる。定義された生成規則が適用されていくと図 4.17が図 4.18のように木構造の描画が行われる。

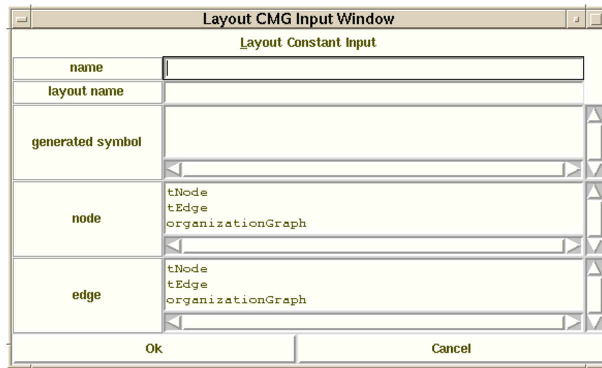

図 4.14: 組織図の軟かいレイアウト CMG 入力ウィンドウ (1)

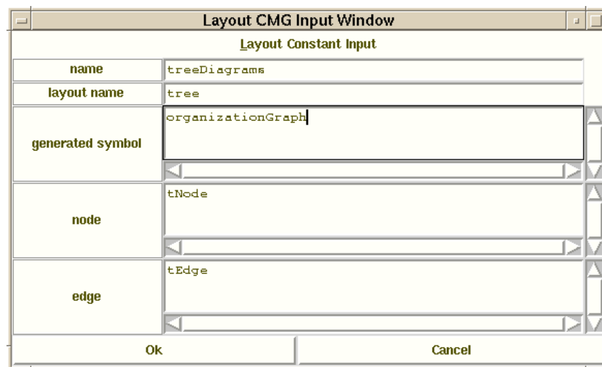

図 4.15: 組織図の軟かいレイアウト CMG 入力ウィンドウ (2)

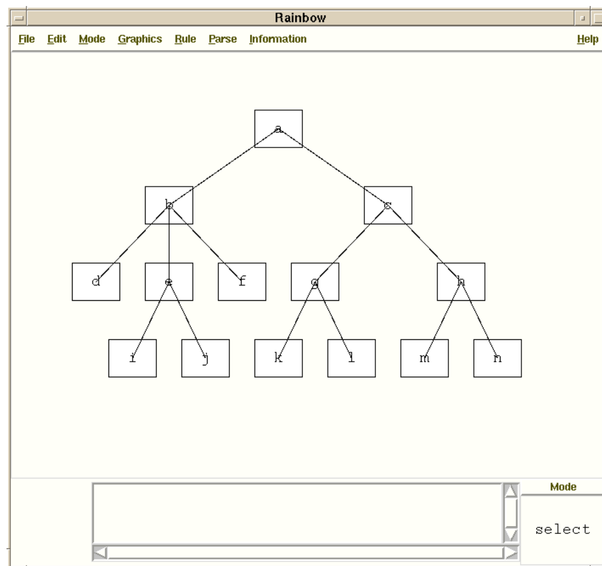

図 4.16: レイアウト後の組織図

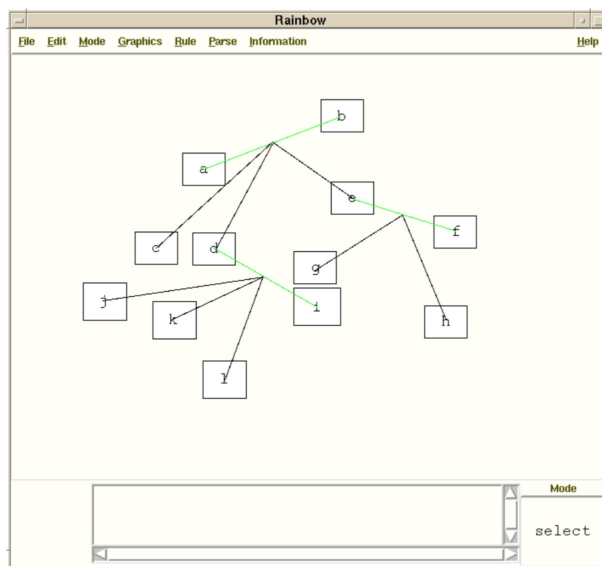

図 4.17: レイアウト前の家系図

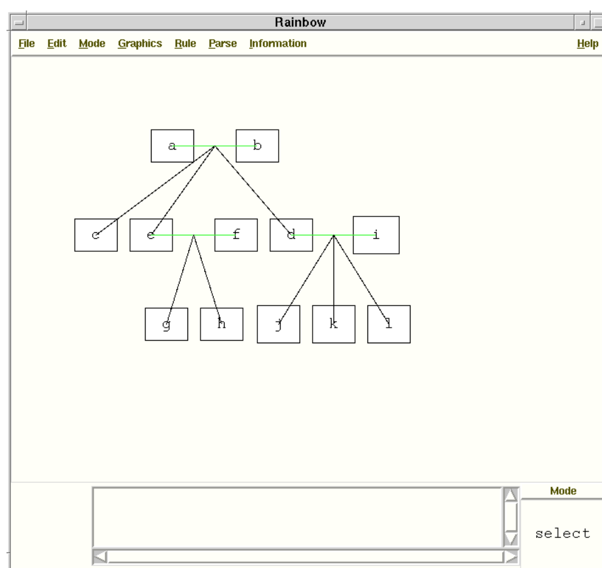

図 4.18: レイアウト後の家系図

第 5 章

「Rainbow」の評価実験

我々が提案したビジュアルシステム生成系「Rainbow」の有効性を明らかにすることを目的として評価実験を行った。

恵比寿で生成されたビジュアルシステムと「Rainbow」で生成されたビジュアルシステムを使い、図形をレイアウトすることを被験者の協力のもとで評価実験を行った。

恵比寿で生成されたビジュアルシステムではレイアウト機能がないので、ユーザが直接図形を描きながらレイアウトを行う。「Rainbow」で生成されたビジュアルシステムでは、図形を解釈するモードとして自動モードと要求モードがある。それで、以下の二つの方法でレイアウトすることが可能である。

- 自動モードでレイアウトする方法
- 要求モードでレイアウトする方法

5.1 実験環境

実験環境のハードウェアは Sun Ultra1 で、 OS は Soralis7 を使用して実験を行った。

被験者は本研究室の 7 人の学生に対して行った。全員マウスの使用には熟知しているが、恵比寿や「Rainbow」の使い方について慣れている被験者と慣れていない被験者がいる。

実験の条件として、評価実験で作成されたグラフは、指定通りのグラフの作成が

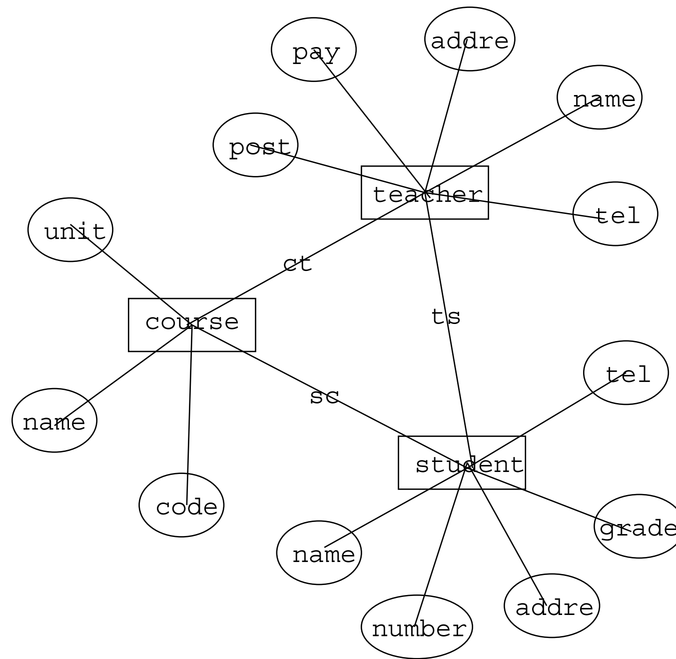

図 5.1: E-R ダイアグラム

完了していることを操作終了の条件とする。被験者がマウスで最初にドラッグすると同時に時間の計測を開始し、被験者がグラフの作成を終了しレイアウトが完了した時点で計測を終了として、作業時間を計る。

5.2 評価実験 1

評価実験 1 は、「Rainbow」によって生成されたビジュアルシステムのレイアウト機能の有効性を明らかにすることを目的としている。

タスクでは、図 5.1のような E-R ダイアグラムと図 5.2のような家系図を用意する。被験者は、恵比寿で生成されたシステムと「Rainbow」で生成されたシステムの両方を使い、用意した E-R ダイアグラムと家系図を見ながら一から図形エディタに描く作業を行う。「Rainbow」で生成されたシステムでは、自動モードもしくは要求モード用いてレイアウトを行うようにする。我々は、レイアウトの作業が終ってから、それぞれについて以下のようなアンケートを被験者に作成してもらった。

質問 1 ビジュアルシステムでは、レイアウト機能が必要だと思いますか？

その理由を書いて下さい。

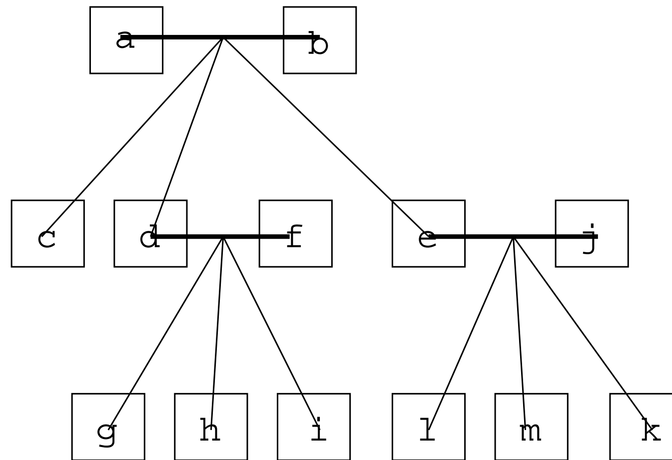

図 5.2: 家系図

質問 2 「Rainbow」で生成されたビジュアルシステムの自動モードと要求モードのレイアウトする方法の中で、どちらがレイアウトしやすいですか？
その理由を書いて下さい。

評価実験 1 の結果は次の通りである。

まず、アンケートの質問 1 において、 E-R ダイアグラムに対する質問 1 の答えは、被験者 7 人全部レイアウト機能が必要だと思っている。その理由は、

- レイアウトを考えずに図形を描くので、ユーザの作業負担を減らすのに役立つからである。
- 適当に図形を描いても見やすくしてくれるからである。
- 複雑な図形を描くときにいちいち手で図形のレイアウトを直すのは面倒だからである。

などである。

また、家系図に対する質問 1 の答えは、被験者 6 人がレイアウト機能が必要だと思っている。その理由は、 E-R ダイアグラムの質問 1 の理由と同じである。しかしながら、被験者 1 人はレイアウト機能が必要ではないと思っている。その理由に

ついては、「家系図は階層的に分かれているのをつなぐだけなので、最初からユーザでもうまくレイアウトできるからである」と述べている。

アンケートの質問 2 において、 E-R ダイアグラムの場合、要求モードがレイアウトしやすいと答えた被験者は 5 人、自動モードがレイアウトしやすいと答えた被験者は 2 人である。家系図の場合は、要求モードがレイアウトしやすいと答えた被験者は 2 人、自動モードがレイアウトしやすいと答えた被験者は 4 人である。ここで、要求モードがレイアウトしやすいと思った理由は、以下になっている。

- 自動モードは、新たな図形の入力があるたびにパーシングとレイアウトを行うので、処理速度が遅いからである（E-R ダイアグラムの場合）。
- 自動モードは、意図してないところに図形が移動することがあるからである。
- 自動モードは、自分のレイアウトイメージが壊れるので、要求モードがいい。

また、自動モードがレイアウトしやすいと思った理由は、以下になっている。

- 家系図の場合は、パーシングとレイアウトの処理速度が早かったのでレイアウトしやすかった。
- 自動モードでは、図形が認識されたかどうかを入力ごとに知ることができるから、安心して入力できる。しかしながら、要求モードでは、図形を全部描いてパーシングするので、もしパーシングされてない図形があると、パーシングが無駄になる可能性がある。
- 自動モードでは、新たな図形を描くときにスペースを作り直す必要がなくて便利である。

E-R ダイアグラムの場合のように、パーシングやレイアウトを行うのに時間がかかる図形は、要求モードでレイアウトする方が好まれる。しかし、家系図のように、パーシングやレイアウトを行うのに時間がかからない図形は、ユーザがレイアウトしやすいと思うモードでレイアウトする方が好まれる。

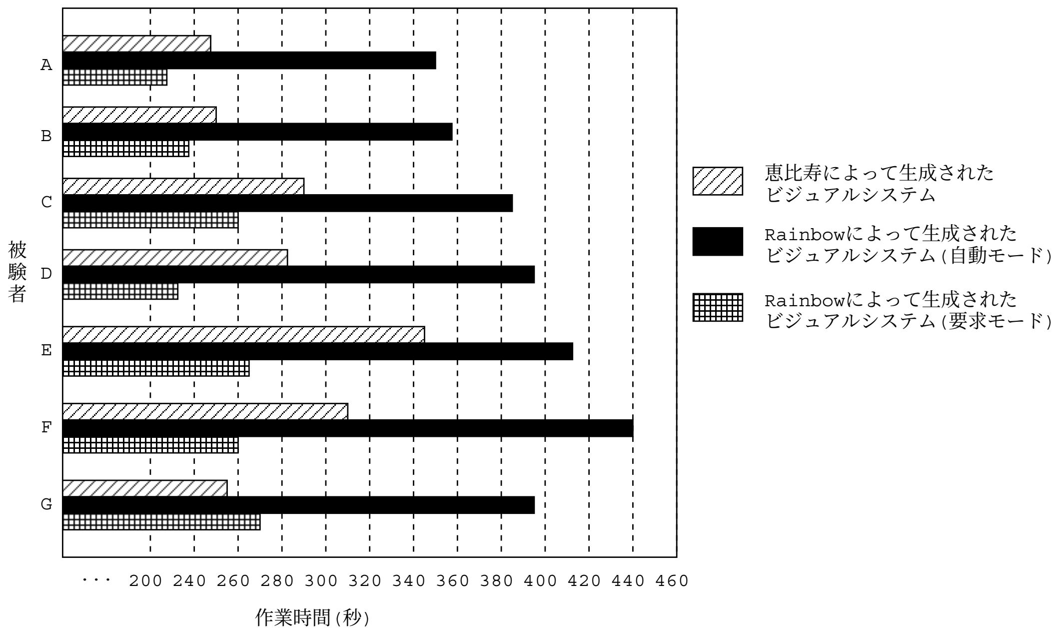

図 5.3: E-R ダイアグラムの実行結果

5.3 評価実験 2

評価実験 2 は、恵比寿で生成されたシステムと「Rainbow」で生成されたシステムの両方を使い、評価実験 1 で用意した E-R ダイアグラムと家系図に対するレイアウトの作業時間を計る。

評価実験 2 の結果を図 5.3と図 5.4に示す。

E-R ダイアグラムの場合、被験者 G を除いて、「Rainbow」によって生成されたビジュアルシステムの要求モードでレイアウトする方が恵比寿で生成されたシステムで手でレイアウトする方より、短い作業時間で図形を作成している。作業時間の平均については、「Rainbow」によって生成されたビジュアルシステム（要求モード）では 249.3 秒、恵比寿によって生成されたビジュアルシステムでは 283.1 秒である。この結果から平均時間で約 12% の作業時間が短縮されていることが分かる。

家系図の場合、「Rainbow」によって生成されたビジュアルシステムの要求モードや自動モードでレイアウトする方が恵比寿で生成されたシステムを用いて手でレイアウトする方より、短い作業時間で図形を作成している。作業時間の平均については、「Rainbow」によって生成されたビジュアルシステムの要求モードでは 120.6

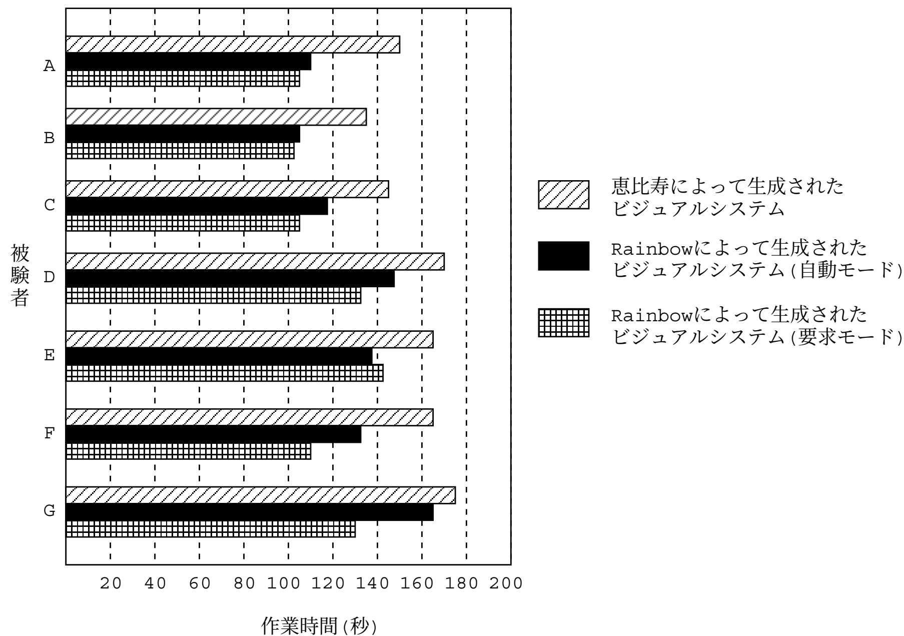

図 5.4: 家系図の実行結果

秒、自動モードでは 131.6 秒、恵比寿によって生成されたビジュアルシステムでは 159 秒である。この結果から平均時間で約 24%（要求モード）、17%（自動モード）の作業時間が短縮されていることが分かる。

次は、「Rainbow」によって生成されたビジュアルシステムの要求モードと自動モードを比べてみる。

E-R ダイアグラムの場合、要求モードの方が自動モードの方より、平均時間で約 36% の作業時間が短縮されていることが分かる。E-R ダイアグラムの場合は、要求モードでレイアウトする方がいい。家系図の場合、要求モードの方が平均時間で約 8% の作業時間が短縮されていることが分かる。

第 6 章

関連研究

6.1 空間パーサ生成系の関連研究

空間パーサ生成系の関連研究としては、SPARGEN[2]、Penguins[3] [4]、恵比寿 [5] [6] [8]、VIC[9] などがある。

SPARGEN は、OOPLG（Object-Oriented Picture Layout Grammars）を用いて図形言語の文法を定義することで空間パーサを生成するシステムである。OOPLG では、図形の属性や制約を C^{++} を用いて定義している。Penguins では、図形の文法を CMG を用いて空間パーサを生成している。しかしながら、これらの空間パーサ生成系は、テキストを用いて図形言語の文法を定義するので、ユーザは文法を理解している必要があり、一般のユーザにとって使いやすいものとはいえないという問題があった。

我々が研究を行っている恵比寿では、ユーザが入力した図形を用いて大まかな図形の CMG の文法を自動的に生成するようにしている。そのあとユーザは、生成された文法を見ながら、制約の追加または削除などをおこない修正する。

恵比寿の図形エディタは、図形言語の文法を定義する定義ウィンドウと実際に図形の解釈を行う実行ウィンドウに分れていたが、「Rainbow」では一つのウィンドウに統合した。なぜなら、ユーザは文法を定義しながら図形を描いて文法の正しさを検査することの繰り返しによって文法を定義していくので、文法の定義モードと図形の実行モードを一緒にする方が便利である。そうすれば、一つの非終端シンボルとしたい図形の中に他の生成規則によって定義された非終端シンボルを認識することができる。また、複雑な図形を表示するのに画面の空間を有効に使える。

さらに、「Rainbow」では恵比寿が扱う制約の他にレイアウト制約を実現している。特に、軟かいレイアウト制約の導入により、図形の全体を解釈しながらインタラクティブにバランスよくレイアウトすることが可能になっている。

VIC では、視覚的な制約入力インターフェイスを恵比寿上に実装し、テキスト編集を行わずに CMG の文法を生成している。

6.2 レイアウトシステムの関連研究

最近、 Marriot らは空間パーサ系 Penguins[12] にレイアウト制約を追加することで図形のレイアウトを行うことを提案している。 Penguins が提供するレイアウト制約は、「Rainbow」の硬いレイアウト制約に相当するものであるが、 Marriot らは、硬いレイアウトの具体的な応用例として二分木、数学の方程式、状態遷移図などを挙げている。

しかしながら、レイアウトでは、図形の全体を把握しやすくバランスよくレイアウトすることが大事であるが、 Penguins では、図形の全体をバランスよく分りやすくレイアウトするための軟らかいレイアウト制約を提供してない。

その他のレイアウトシステムとして TRIP[27] がある。 TRIP とは、テキスト形式の抽象オブジェクトとその関係からレイアウトされた図形を生成するシステムである。抽象オブジェクトとその関係はユーザが定義するマッピング規則により、図形オブジェクトとその関係に変換され、レイアウトシステムより図形が生成される。マッピング規則は、ユーザが Prolog で記述している。 TRIP では、制約に基づいて図形要素の位置が決められてから図形が生成されるが、そのあと図形要素間に制約が課せられないので、制約を保つつつ図形を動的に編集することができない。 TRIP でもグラフレイアウトシステムが存在し、無向グラフの描画アルゴリズムを使って図形のレイアウトを行うが、図形要素の位置を決める図形関係と混じってレイアウトすることはできない。

第 7 章

結論

本論文では、ビジュアルシステム生成系にレイアウト制約を扱えるようにして、図形をバランスよく見やすくレイアウトできるビジュアルシステム生成系の研究について述べた。

レイアウト制約は、軟かいレイアウト制約として、スプリングモデル制約、マグネティックスプリングモデル制約、リスト構造制約、木構造制約などを導入した。ここで、スプリングモデル制約は無向グラフのレイアウトを行う場合に用いる。マグネティックスプリングモデル制約は、エッジの方向を考えて有向グラフのレイアウトを行いたいときに用いる。また、リスト構造制約と木構造制約は、それぞれグラフをリスト構造と木構造にレイアウトする場合に用いる。

また、硬いレイアウト制約は、図形の座標や図形間の距離などの制約を具体的に与えたい場合に用いる。

我々は、これらに基づいてビジュアルシステム生成系「Rainbow」を開発した。「Rainbow」によるビジュアルシステム作成の例として、データベース分野で実世界のデータ構造を記述するのに用いられる「E-R ダイアグラム」とオブジェクト指向に基づくソフトウェア設計に用いられる「オブジェクト図」、会社などの仕組みを表すのに用いられる「組織図」、親族の関係を表すのに用いられる「家系図」の例を示した。

本論文の新規性は以下の通りである。

- **CMG の一つの生成規則**として軟かいレイアウト制約を導入し、図形の全体をグローバルにレイアウトすることを提案、実装した点。

- 硬いレイアウト制約と軟かいレイアウト制約を混ぜて用いることにより、適用できるビジュアルシステムの応用範囲を広げた点。
- 空間パーサにレイアウト制約を追加することにより、パーシング中に図形を自動描画することで図形をよりインタラクティブに処理することを可能とした点。

これからの研究の課題として以下の点を挙げる。

一つ目は、直接操作の適用に関してである。「Rainbow」ではユーザが入力した図形を用いて大まかな図形の CMG の文法を自動的に生成し、そのあとテキスト編集を行って文法を定義するが、我々が研究を行っている VIC と統合することにより、もっとインタラクティブに直接操作を用いて図形言語の定義が可能になると考えられる。

二つ目は、制約の強さに関しての課題が挙げられる。現在の「Rainbow」では、通常の制約と硬いレイアウト制約を同じ強さで扱っているが、今後は制約階層の導入が考えられる。制約階層とは、強さと呼ばれる制約の優先度を用いて、強い制約をできるだけ多く満し、より弱い矛盾する制約を無視するように制約系の解を求める方法である。通常の制約を硬いレイアウト制約より強い制約として定義することにより、硬いレイアウト制約を起動することで、以前に充足していた通常の制約が満されなくなる場合の問題を解決することができると考えられる。

三つ目は、三次元のビジュアルシステムへの適用に関してである。我々の研究室では、三次元空間に図形の視覚化により大規模な図形への対応、リアルな表現、レイアウトの自由度拡大という利点のため三次元ビジュアルプログラミングシステム「3DPP」 [28] [29] の研究を行ってきた。また、三次元オブジェクトの操作手法に自動グラフレイアウト機能を適用し、図形の可読性を向上させる研究を行ってきた [30]。これらの研究に基づいて、図形を三次元空間で扱うビジュアルシステムへの拡張が考えられる。

四つ目は、図形の構成要素が増えるとパーシングするのに時間がかかるので、「Rainbow」の解釈アルゴリズムの効率化が挙げられる。

参考文献

[1] 杉山公造: グラフ自動描画法とその応用 - ビジュアル ヒューマン インタフェース -, 計測自動制御学会, (1993).

[2] Eric J. Golin and Tom Magliery: A Compiler Generator for Visual Languages, *Proceedings of the IEEE Symposium on Visual Languages*, pp. 314–321, (1993).

[3] Sitt Sen Chok and Kim Marriott: Automatic Construction of User Interfaces from Constraint Multiset Grammars, *Proceedings of the IEEE Workshop on Visual Languages*, pp. 242–249 (1995).

[4] Sitt Sen Chok and Kim Marriott: Automatic Construction of Intelligent Diagram Editors, *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp. 185–194 (1998).

[5] 馬場昭宏, 田中二郎: Spatial Parser Generator を持ったビジュアルシステム, 情報処理学会論文誌, Vol. 39, No. 5, pp. 1385-1394 (1998).

[6] 馬場昭宏, 田中二郎: 「恵比寿」を用いたビジュアルシステムの作成, 情報処理学会論文誌, Vol. 40, No. 2, pp. 497-506 (1999).

[7] Kim Marriott: Constraint Multiset Grammars, *Proceedings of the IEEE Symposium on Visual Languages*, pp. 118–125 (1994).

[8] Akihiro Baba and Jiro Tanaka: Eviss : a Visual System Having a Spatial Parser Generator, *Proceedings of Asia Pacific Computer Human Interaction*, pp. 158–164 (1998).

[9] Kenichirou Fujiyama, Kazuhisa Iizuka and Jiro Tanaka: VIC:CMG Input System Using Example Figures, *Proceedings of the International Symposium on Future Software Technology*, pp. 67–72, (1999).

[10] 丁 錫泰, 田中二郎: Rainbow: ビジュアルシステム生成系におけるレイアウト制約の実現, 情報処理学会論文誌, Vol. 41, No. 5, pp. 1246-1256 (2000).

[11] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen: *Object-Oriented Modeling and Design*, Prentice-Hall International (1991).

[12] Sitt Sen Chok, Kim Marriott and Tom Paton: Constraint-based Diagram Beautification, *Proceedings of the IEEE Workshop on Visual Languages*, pp. 12–19 (1999).

[13] Jauhar Ali and Jiro Tanaka: Automatic Code Generation from the OMT-based Dynamic Model, *Proceedings 2nd World Conference on Integrated Design and Process Technology*, pp. 407–414 (1996).

[14] Jauhar Ali and Jiro Tanaka: Generating Executable Code from the Dynamin Model of OMT with Concurrency, *Proceedings IASTED International Conference on Software Engineering*, pp. 291–297 (1997).

[15] アリ ジョハル, 田中二郎: オブジェクト指向方法論（OMT）に基づく動的モデルからの Java コード生成, 情報処理学会論文誌, Vol. 39, No. 11, pp. 497-506 (1998).

[16] Suchtae Joung, Jauhar Ali and Jiro Tanaka: Automatic Animation from the Requirement Specification based on Object Modeling Technique, *Proceedings International Symposium on Future Software Technology (ISFST-97)*, pp. 133-139, (1997).

[17] Suchtae Joung and Jiro Tanaka: Icon-based Animation from the Object and Dynamic Models based on OMT, *Proceedings Asia Pacific Computer Human Interaction*, pp. 133-139, (1998).

[18] Satoshi Nakashima, Jauhar Ali and Jiro Tanaka: An Automatic Layout System for OMT-based Object Diagram, *Proceedings of the Second World Conference on Integrated Design and Process Technology 1996 (IDPT'96)*, pp. 82-89, (1996).

[19] 野口隆佳: インタラクティブ性を考慮したオブジェクト図の自動レイアウト手法, 筑波大学工学研究科修士論文 (2000).

[20] Peter Eades: A Heuristic for Graph Drawing, Congressus Numerantium, Vol. 42, pp. 149-160 (1984).

[21] 三末和男, 杉山公造: マグネティック・スプリング・モデルによるグラフ描画法について, 情報処理学会研究報告 ヒューマンインタフェース 55-3, pp. 17–24 (1994).

[22] John Q. Walker: A Node-positioning Algorithm for General Trees, Software Practice and Experience, Vol. 20, No. 7, pp. 685-705, (1990).

[23] Michael Sannella: Constraint Satisfaction and Debugging for Interactive User Intefaces, Technical report, University of Wasington (1994).

[24] Peter Chen: The Entity-Relationship Model: Towards a Unified View of Data, *ACM Transactions Database System*, Vol. 1, No. 1, pp. 9-36 (1976).

[25] 中島哲, 田中二郎: オブジェクト指向方法論に基づくオブジェクト図の自動レイアウト, 情報処理学会論文誌, Vol. 39, No. 12, pp. 3282-3293 (1998).

[26] Takayoshi Noguchi and Jiro Tanaka: New Automatic Layout Method based on Magnetic Spring Model for Object Diagrams of OMT, *Proceedings of Future Software Technology*, pp. 89–94 (1998).

[27] Tomihisa Kamada and Satoru Kawai: A General Framework for Visualizing Abstract Objects and Relations, *ACM Transactions on Graphics*, Vol. 10, No. 1, pp. 1–39 (1991).

[28] 宮城幸司, 大芝崇, 田中二郎: 三次元ビジュアル・プログラミング・システム 3D-PP, 日本ソフトウェア科学会第 15 回大会論文集, pp. 125-128, (1998).

[29] 田中二郎: 3 次元ビジュアルプログラミングシステム 3DPP の構築に向けて, 人工知能基礎論研究会 (第 34 回), SIG-FAI-9802, pp. 9-14, (1998).

[30] 宮下貴史: 三次元ビジュアルプログラム編集環境の構築, 筑波大学理工学研究科修士論文 (2000).

謝辞

本論文をまとめるにあたり、終始、親身なご指導を頂いた、主任指導教官である筑波大学電子・情報工学系教授 田中二郎博士に深く感謝致します。また、本論文について有益なご助言を頂きました筑波大学電子・情報工学系教授 西川博昭博士、同教授 西原清一博士、同教授 福井幸男博士、同助教授 山本幹雄博士に深く感謝申し上げます。

本論文のシステムの基になった恵比寿を作成した馬場昭宏氏からは、本論文について貴重なコメントを頂きました。本当に心より感謝致します。また、恵比寿の研究に共同で取り組んだ恵比寿グループのメンバーである飯塚和久氏、藤山健一郎氏とは、システムの研究・開発にあたり有益な議論をしました。ここに感謝の意を表します。さらに、インタラクティブプログラミング研究室 (iplab) の各員にも深く感謝致します。