ソフトウェア基礎論　16－10
プログラミング言語　4－10
（１９８６．２．７）

属性伝搬によるコンストレイントの実現

松田裕幸　（日本電気技術情報システム開発（株））

概要　コンストレイント（制約）指向プログラミング(constraint-oriented programming) を属性の伝搬によって実現するアイデアについて提案する。一般に、プログラム仕様は制約を有効に用いることで仕様の記述が容易になり、記述量も大幅に減少する。また、理解もしやすくなる。本稿では、制約を用いたプログラミングの例を示すことで、属性の伝搬が制約条件の満足に与える役割について述べ、そのために属性文法を拡張する場合の条件についても議論する。

１．　はじめに

属性文法(attribute grammar) はプログラミング言語のシンタックスとセマンティックスを数学的に定義するためKnuth によって発案され、従来は、プログラミング言語の仕様記述言語としての側面と、コンパイラ生成系を通して生成されるコンパイラの中に埋め込まれた属性評価機(attribute evaluator) の側面に研究が集中してきた。しかし、現在では、属性文法は通常のプログラミング仕様やシステム仕様の記述言語としても利用され始めている。例えば、構造エディタの仕様定義に用いられていることはよく知られている[2,3] 。また、属性文法を用いていることを明示していないが属性文法の概念を利用しているシステムまで含めると相当数のシステムが対象となるはずである。実際、用いるデータ構造が木構造をなし、かつ、各ノードが所有する情報（属性）に対して操作を施したり、情報を他のノードに伝えたりするようなシステムにおいては、原理的には、属性文法の考え方が利用できる。

プログラム仕様を記述する言語として属性文法を取り上げたときの長所としては、1)高いモジュール性、2)属性変数の作用的働き（値の書き換えを許さない）による仕様の妥当性検査の容易さ、等を挙げることができる。しかしながら、実際の記述においては、これらの長所をもってしても克服しがたい問題が存在する。すなわち、プログラム仕様がいかに宣言的に記述され、また、いかにモジュール階層がきれいに整理されていても、仕様の大部分は仕様中で用いる変数への操作であり、その記述がアドホックになっていると、全体的な長所が失われることになる。これは、オブジェクト指向言語においても同様である。オブジェクト指向の場合は各オブジェクトの内部変数が今ここで議論している変数にあたるが、複数のオブジェクトに分散する変数へのアクセス（概念上はメッセージを通して行なう）は処理効率を下げるだけでなく、仕様の理解の妨げになる。

従って、仕様記述で用いられる変数への操作を整理することが重要になってくる。本稿では、ThingLab[1] 等で提案されているコンストレイント（制約）指向プログラミングの概念を、属性文法をベースにした仕様記述言語に利用することを考える。制約によって複数の変数間の値（属性値）の関係を

規定でき、また、この関係を保持しなければならない条件から制約の伝搬が発生する（属性の伝搬によって実現される）。制約の伝搬は次々に他の変数の変更等を行ない、ある関係が作られている変数間同士の操作を統一的に行なうことを可能にする。第２節では、コンストレイント指向プログラミングの具体例を挙げるとともにその記述例も示す。第３節では、属性文法で制約を記述する場合生じる基本定義からの逸脱について議論する。

２．　コンストレイント指向プログラミング

2.1 例と記述

【例１】　線分の定義と操作

いま、グラフィック図形の表示と操作のための仕様を考えてみる。簡単のために、線分を取り上げる。線分は２点からなるとすると、その記述は以下のようになる。

(2.1)

```
Line := Point Point
  Line.place <- Point$1.place
  Line.start <- Point$1.place
  Line.end <- Point$2.place
  Line.size <- distance(Point$1.place, Point$2.place)
```

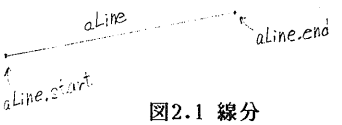


図2.1 線分

線分Lineは２つの点Point からなり、線分の始点Line.startは点１の位置Point$1.place で得られ、線分の終点Line.endは点２の位置Point$2.place で得られる。線分の長さは２点間の距離で与えられ、線分の位置は、一応、点１の位置で定義しておく。記号:=は構成関係（あるいは部分－全体関係）を表わす。この例では、「LineはPoint とPoint から成る」と読む（右辺の項の順番については特に規定していないことに注意）。また、ドットで区切られた左側は対象（あるいは、オブジェクト、成分、ノード）を表わし、右側は対象の属性を表わす。ただし、$は同じ対象名のものを区別するために用いている。例えば、Point$1 は上記定義の右辺の第１要素を表わす。対象Lineの属性としてはここでは、start とend がある。記号<-は属性への値の代入(attribution) を表わす。関数distanceは属性上の計算を行なうために用いる補助的な関数（属性関数）である。属性start, end, place はすべて同一の型を持つ：

```
Place = Real x Real
start, end, place : Place
```

次に線分への操作を記述する。操作としては、線分の端点の一方を固定して他の端点を移動する線分の拡大、縮小、あるいは、移動がある。図(2.2a, 2.2b)で示すと、

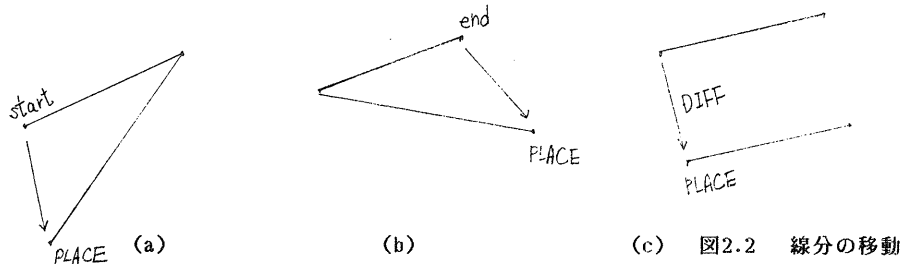


図2.2　線分の移動

もうひとつの操作としては、線分そのものの移動がある（図2.2c）。

これらの操作を記述するために(2.1) の仕様に以下のものを追加する。

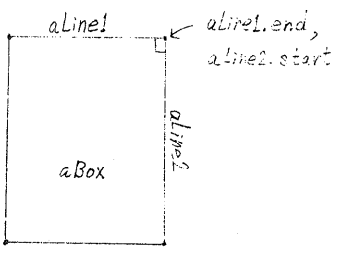
```
(2.2)
messages:
  move_start_point PLACE : Point$1.place <- PLACE
  move_end_point PLACE : Point$2.place <- PLACE
  move DIFF : Point$1.place <- Point$1.place + DIFF
              Point$2.place <- Point$2.place + DIFF
```

対象に対する操作はメッセージで指定する。メッセージ"move_start_point PLACE"によって端点１の位置Point$1.place がPLACE に変更される。move_start_pointはメッセージタグで、PLACE は変数である。線分自体を移動するためのメッセージは"move DIFF" である。DIFFとPLACE は図2.2 に示した通りである。

【例２】　箱の定義と操作

次にいま定義した線分を用いて箱(Box) を定義する（図2.3)。また、操作の例を図2.4 に示す。

```
(2.3)
Box := Line Line Line Line
  Box.place <- Line$1.place
  Box.size <- mult(Line$1.length, Line$2.length)
  constraints:
    Line$1.end <=> Line$2.start
    Line$2.end <=> Line$3.start
    Line$3.end <=> Line$4.start
    Line$4.end <=> Line$1.start
    cross(Line$1, Line$2) = 90
```



```
    cross(Line$2, Line$3) = 90
    cross(Line$3, Line$4) = 90
  messages:
    move PLACE : Line$1.place <- PLACE
    expand SCALE : Line$1.end <- Line$1.end + mult(Line$1.length, SCALE)
    stretch LINE PLACE : LINE.place <- PLACE
```

図2.3 箱の定義

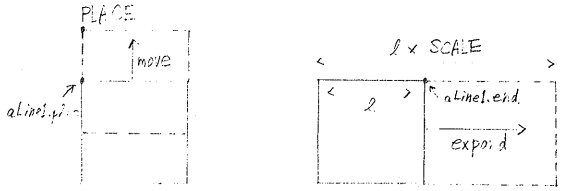


図2.4 箱への操作

箱は４つの線分からなる。箱の位置と大きさはそれぞれ(2.3)の２行目、３行目で与えられる。４行目以降は、コンストレイント（制約constraint）を表現する。ここで、記号<=> と＝は異なった役割を持つ。まず、＝は左辺と右辺が等しいことを制約する。例えば、

```
cross(Line$1, Line$2) = 90
```

は、線分１と線分２のなす角度が90度でなければならないことを表現する。一方、

```
Line$1.end <=> Line$2.start
```

は、線分１の終点と線分２の始点が同じであることを制約すると共に、線分１の終点が変化したら線分２の始点が同時に変化しなくてはならないことを表わしている。線分２の属性値(end)が変化すると線分１の属性値(start)も制約によって変更される。一方の変化を他方に反映させるようなタイプの制約を実現しているのが、属性の伝搬機構である。属性の伝搬は仕様には明示的に記述されておらず、属性値の変化によってデモンが働き、暗黙の伝搬が起こる。例えば、線分１の変化を線分２へ伝えることを表わす以下の記述について見てみる。

```
aLine2.start move_start_point aLine1.end
```

これは線分２に対して、メッセージタグmove_start_pointと共に線分１の終端位置を変更すべき位置として送ることを表わしている。ここで、aLine1,aLine2 はそれぞれLine1,Line2 のインスタンスである。属性文法では属性の値の決定は静的であって、普通の変数のように再代入を許していないが、ここでは、属性の値は変更を許している。しかし、これは無制限ではなく一定の制約のもとで行なわ

れる。例えば、線分２の始点は固定であり、線分１の変化を受けたくないとするときは、

```
aLine2.start <- (fix) aLine2.start
```

のようにする。これは最初の値の決定後は変更を許さないことを宣言する。

箱のインスタンスを一つ作りaBoxとする。その箱に対する移動は箱に対して次のメッセージを送るだけでよい。

```
aBox move PLACE
```

これによって、Line2 のインスタンスaLine2の位置が変化し、これが線分２と制約によって関係付けられた全ての線分のインスタンスの属性を変更する。しかし、制約を伝播させるためには、仕様(1.3)のメッセージmoveの記述に問題がある。すなわち、線分１の位置を変更しても制約で関係付けた始点と終点の位置を変えることはできない。理由はLineの定義にある。線分の始点Line.startと終点Line.endの値は点の位置のみによってしか定義されていない。よって、次のものを追加しないと属性の伝搬を引き起こすことはできない。

```
Line.start <- Line.place
Line.start <- Line.end - Line.size
Line.end <- Line.start + Line.size
```

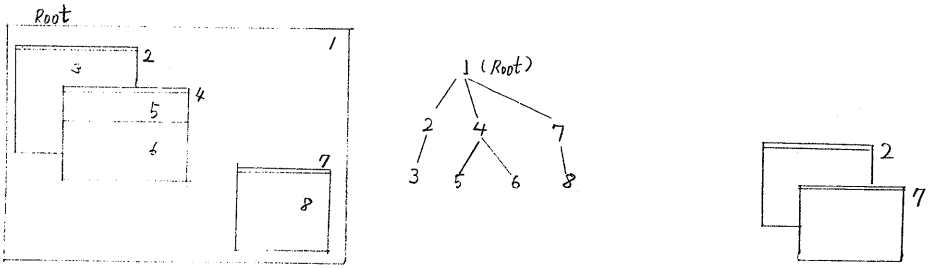
箱の拡大、部分の拡張はそれぞれメッセージexpand, stretch によって定義される。そして、移動の場合と同様、属性値の変化に対しては制約を満足するような属性の伝搬が起きる。

【例３】　ウインドウ管理データベース

最後の例としてウインドウ管理のための簡単なデータベースについて考えてみる。話を簡単にするために次の仮定を設ける。

1) 全てのウインドウはルートウインドウ（画面全体）の子供として定義される。
2) 各ウインドウ内のサブウインドウ同士は重なり合わない。（よって、ウインドウの階層の深さは最大３レベルである。）
3) ウインドウの階層において同一レベルでは左側ほど年齢が高く、右側ほど若い。

図2.5aに現われているウインドウの表示に対するウインドウの階層関係を図2.5bに示す。番号２のウインドウと番号７のウインドウは画面において重なり合っていないが、階層関係を基にすると一方を移動して他方に重ねたときは図2.5cのようになる。



(a)　　(b)　　(c)　　図2.5 ウインドウの階層関係

ウインドウ階層が変化する場合は、ウインドウを新しく作成する(create)、削除する(delete)、選択する(select)する、ハイドする(hide)場合のみである。階層構造の変更の規則は次の通りである：

1) ウインドウの作成、選択においてはそのウインドウの年齢を最も若くする。ただし、他のウインドウの年齢関係の順序は保存される。

2) ウインドウのハイドではそのウインドウの年齢を最も高くする。

3) ウインドウを削除したときは階層から取り除く。

それぞれの操作を容易に記述するために例２同様、制約を利用する。

(2.3)

```
Root := Window Window ... Window
  Root.youngest_son : the youngest son of Root tree
  Root.oldest_son   : the oldest son of Root tree
  ...
  constraints:
    Root.oldest_son.older = nil
    Root.youngest_son.younger = nil
    Window.older.age - Window.age = 1
    Window.age - Window.younger.age = 1
    over(Window1, Window2) <=> Window1.age < Window2.age
    under(Window1, Window2) <=> Window1.age > Window2.age
    ...
  messages:
    select WINDOW : WINDOW.older.younger <- WINDOW.younger
                    Root.youngest_son <- WINDOW
    move WINDOW PLACE : WINDOW move PLACE
    ...
```

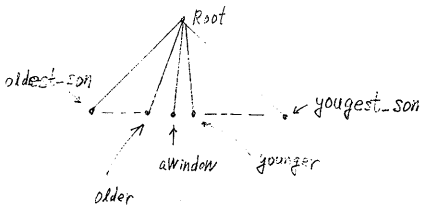


図2.6 ウインドウの属性

紙面の都合上、この仕様の細かい説明は省略する。また、制約条件として不足している部分もある。例として、今、あるウインドウWINDOWを選択する(select)場合を考える。まず、ウインドウの兄弟関

係を変更する。次に、ルートの最年小ウインドウとして選択したウインドウを指定する。これらの変更に対して制約によってウインドウの一部の属性値が決定され、次に属性の伝搬によって関連する他のノードの属性の変更を起こす。

2.2 コンストレイント指向プログラミング

プログラム仕様を記述する際に制約を用いることの利点は、次の通りである：

1) 局所的な関係を記述するだけで、全体の制御を管理できる。
2) 通常の手続きで記述するのと比較してかなり少ない記述量ですむ。
3) 1),2) の理由からプログラムの論理構造の見通しが良くなる。

1)の性質が好ましい状況はよく起きる。仕様の一部を記述している時点では全体が見えず、やむおえず自分と周辺との関係のみを記述しておく場合などである。また、一般に全体が見えている場合でも記述は常に局所的な部分に制限されているために理解を容易にする。プログラムの独立管理はモジュールの概念やオブジェクト指向のモデルでも実現できるが、これらと制約指向の決定的違いは各オブジェクトもしくはモジュールの内部構造と対象間の関係の設計にある。例えば、オブジェクト指向ではオブジェクトの切り出しはうまくできたとしても、ある操作を実現するためにオブジェクト内の変数への多くの操作を記述し、かつオブジェクト間のメッセージのやりとりを細かく制御するプログラムを書かなくてはならない。しかし、この操作は実はある制約を実行したいだけかもしれない。この時、変数への参照、補助的なメッセージ呼び出し等の記述があるために本当に実現したい部分が見えなくなる恐れがある。

2)は制約指向プログラミングの実現とも絡んでくる話である。基本的には制約関係を保持するために暗黙の手続き（デモン）が起動され、プログラム中に明示的に表現しなければならない記述を減らせる点に特長がある。

制約の役割には次の３つの働きがある。すなわち、

1) ある値は「．．．でなければならない」という制約。これは２節のconstraints では記号＝で表現されていたものである。
2) 1)の制約が崩れたときはその原因（具体的には属性値）を制約の他方に伝播する。これは、記号<=> で表現されていた。
3) 制約の伝搬を常に起こしては当然困る場合がある。その時には伝搬を制限しなくてはならない。これは記号(fix) で表現されていた。

本稿は制約指向プログラミングを直接の話題としている訳ではないので、ここではこれ以上の議論

は行なわない。

３．　属性文法からの逸脱と拡張

２節で紹介した属性に関する考え方の中には、属性文法から逸脱した概念がいくつか見られる。本節では、属性文法の言語設計以外への応用の観点から、それらの逸脱の必要性を述べる。逸脱の主なものとしては、次の３点をあげることができる。

1)　属性付き木構造の任意のノードから任意の方向へ属性を伝搬できる。
2)　属性付き木の構造を動的に変更できる。
3)　属性の値の変更が可能。

属性付き木は言語設計のフェーズでは、通常、属性付き（構文）解析木に相当する。属性付き解析木は上昇型、あるいは下降型の構文解析の手法に従って作成され、意味解析時に文脈に関する意味チェックと中間コードの生成を行なうためにルートからトップダウンに各ノードの属性の評価を行なっていく。よって属性評価機の解析木に対する働きは一定している。

しかし、属性付き木は構文解析木としてだけでなくもっと広い使い道がある。木構造という最低限の制限はあるものの、各ノード間（必ずしも近接するノード間だけでなく）の属性の伝搬の制約がある程度緩和されているだけでもかなり応用範囲は広がる。第２節であげた例は上記の事態を全て含んでいるが、これ以外にも、最近よく取り上げられる属性文法の応用として構造エディタの設計がある。

構造エディタが対象とする世界では、プログラムが解析された状態で木構造として常に管理されていて、ビュー（プログラム）の変化に対応して内部構造（木構造）が変化し、場合によってはさらにビューへの変化が生じる。プログラムテキストが修飾される（例えば、削除、追加等）度に対応する内部構造が変えられ、また、変数の宣言を変更する度に対応する変数の使われている文脈のチェックを行なわれる。ここでは、修飾が発生した場所（ノード）から必要な情報（属性値）を関連するノードに伝播させ、その結果、属性付き木の一部が動的に変化させられる。さらに、この過程で属性の値の再代入は何度も生じる。

本節では属性文法を以上のように拡張する際の問題点について、なんら理論的な議論を提出していないが、属性文法が本来持っている能力が少しでも有効に利用できる局面を見いだせたのではないかと考えている。

なお、文献[2] では、属性付き木の各ノードをルールに対応させ、かつノードを一つのオブジェクトとみなし、属性の伝搬はメッセージとする考え方を提出している。属性をメッセージで表わすことで、たとえば、あるルールの属性の個数を変更したい時などは全ルールの再コンパイルは不要となり、変更したルールのコンパイルのみを行なえばよい。また、文献[3] では構造エディタを設計する際に

用いたコッドの関係モデルの不備を属性文法で補う（属性の型として関係relationを導入する）場合の問題点について論じている。

4．　さいごに

コンストレイント指向プログラミングを属性の伝搬によって実現する場合の例を紹介し、その為に属性文法の考え方を拡大する場合の問題点に若干触れた。実際には、制約の伝搬を属性の伝搬で実現するには処理効率を考慮したプログラム仕様のコンパイルが必要であり、また、属性文法との関連も殆ど議論できなかった。しかしながら、属性文法の基本的アイデアは十分広い応用分野を持つことができるはずである。

文献

[1] Borning, A.: The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory., ACM TOPLAS, Vol 3.,No. 4, 1981, 353-387.

[2] Demers, Rogers, Zadeck : Attribute Propagation by Message Passing., ACM SIGPLAN '85 Symposium on Language Issues in Programming Environment., 43-59.

[3] Horowitz, Teitelbaum : Relations and Attributes; A Symbiotic Basis for Editing Environments., ACM SIGPLAN '85 Symposium on Language Issues in Programming Environment., 93-106.