図形言語処理システムにおける 図形エディタと空間解析器の統合

Integration of Diagram Editor and Spatial Parser for Visual Language System

飯塚 和久 † 志築 文太郎 †† 田中 二郎 ††
Kazuhisa IIZUKA Buntarou SHIZUKI Jiro TANAKA

† 筑波大学大学院 工学研究科
Doctoral Program in Engineering, University of Tsukuba
†† 筑波大学 電子・情報工学系
Institute of Information Sciences and Electronics, University of Tsukuba
{iizuka,shizuki,jiro}@iplab.is.tsukuba.ac.jp

図形言語処理システムは，図形言語を文法に従って解析するシステムである。多くのシステムでは図形エディタを用いて図形を編集し，そのデータを空間解析器で解析していた．我々は，インタラクティブな図形言語処理システムを記述するため，図形エディタと空間解析器の統合を行う．この統合のために，図形文法を拡張し，さらに，図形編集機能を図形文法で記述する手法について提案を行い，図形編集機能の具体的な記述例を示す．これにより，図形文法に従った編集を可能とする，インタラクティブな図形言語処理システムを実現した．

1 はじめに

図形言語とは，四角や円，アイコン，文字など，基本的な図形要素を組み合わせて，意味のある図形群を構築できるような図形の集合である．状態遷移図，OMT のオブジェクト図，回路図，数式などは，図形言語として定義できる．図形言語は，文法によって記述することができ，これを図形文法と呼ぶ．図形文法としては，Positional Grammars[1], Relational Grammars[2], Constraint Multiset Grammars (CMG)[3]など，いくつか提案されている．

これらの図形文法に基づいて図形言語を処理するシステムは，図形エディタと空間解析器で構成される．図形エディタでは，図形言語に応じた終端記号にあたる図形を記述する．空間解析器は，図形エディタで記述された図形群を，与えられた図形文法に基づいて構文解析と意味解析を行い，解析結果を出力する．SPARGEN[4]や，Balt らのシステム[5]などの多くのシステムでは，このように図形エディタと空間解析器を別々に扱っている．一方，Eviss[6]や VIC[7], Penguins[8]などは，図形エディタと空間解析器を一つのシステムとして提供している．このことにより，図形を描画すると同時に解析を行い，その出力結果を即座に得られるようになっている．しかし，これらのシステムでは，図形エディタ部が，一般化された処理しか行えないため，図形言語に依存した特別な処理などが行えなかった．また，解析結果に基づいた図形の編集機能を提供することができない．我々は，これらを改善し，拡張性の高い記述が可能な，よりインタラクティブな図形言語処理システムを提案する．

2 図形言語処理システム

2.1 システム構成

我々の図形言語処理システムは，ユーザとのインタラクションを行う入出力部と，図形言語を文法に基づいて処理する空間解析部から成り立つ．入出力部は，図形を描画・編集するためのキャンバスに加え，図形言語ごとにメニューやボタンなどを付加することができる．空間解析部は，あらかじめ与えられた図形文法を基に解析を行う．また，解析結果に応じて図形のレイアウトなどを処理するための制約解消系を利用することもできるようになっている．システムの起動時には，図形文法を読み込むとともに，キャンバスを初期化し，必要に応じてメニューやボタンを作成して利用する．

2.2 図形文法の拡張

我々の図形言語処理システムでは CMG を拡張した図形文法を用いる．これは，CMG ではインクリメンタルな空間解析器で処理することができるため，ユーザの入力によって逐次与えられる図形の変更を解析することが

できるからである. Positional Grammars や Relational Grammars などは, その文法上の制約から, インクリメンタルな解析器を構築できない.

我々の提案する図形文法の生成規則は, 以下のようになる.

```
P ::= P1,…,Pn where (
  Constraint
) {
  Assignments
} {
  Construct_script
} {
  Finalize_script
}
```

この生成規則は, RHS の記号 $P_1,\ldots,P_n$ が *Constraint* を満たしたときに, LHS の P から生成され, その属性値は *Assignments* を基に計算されることを示している.

CMG と異なるのは, *Construct_script*, *Finalize_script* である. *Construct_script* は, この規則が適用されたときに実行されるスクリプトを示している. また, *Finalize_script* は, この規則が適用されて生成された LHS の非終端記号が, 削除された場合に実行されるスクリプトである. なお, 非終端記号の削除は, RHS の記号が削除されるか, 属性が変更されて *Constraint* が満たされなくなった場合に行われる.

例えば, Circle と Text からなる Node という非終端記号を表す生成規則は次のようになる. なお, Node として認識されると, Circle の内部の色が gray に変更されるものとする.

```
 1:N:Node ::= C:Circle, T:Text where (
 2:  C.mid == T.mid
 3:) {
 4:  N.text = T.text
 5:  N.mid = T.mid
 6:} {
 7:  N.prev_color = C.inner_color
 8:  C.inner_color = gray
 9:} {
10:  C.inner_color = N.prev_color
11:}
```

1 行目は LHS の Node から RHS の Circle と Text が生成されることを示している. 2 行目でこの規則が適用される制約条件を示しており, ここでは Circle と Text の中心が一致していることを表している. 4-5 行目では, LHS の Node の属性を決定している. 7-8 行目は, *Construct_script* を示しており, Circle 内部の色を変更している. 10 行目は, *Finalize_script* を示しており, 非終端記号 Node が削除された場合に, Circle の内部の色を元に戻すようになっている.

2.3 図形編集機能の図形文法への統合

これまでのシステムでは, 図形言語に依存しない一般的な図形エディタを用いて図形を編集していた. しかし, 図形編集機能は, 利用する図形言語に応じて変化しうる. これを, 図形文法によって記述することにより, 図形の解析だけに用いられてきた文法と解析を編集機能の記述にも適用し, これらを同一の枠組みで利用できるようにする. つまり, マウスの動きや, メニューなどを図形言語の枠組で扱う. 我々は, このために, 特別な終端記号を導入する. 一般に, 図形言語で用いられる終端記号は, 四角や円, 文字といったオブジェクトを表現する為に用いられるが, マウスやメニューについても終端記号として扱えるようにし, これらの終端記号の属性値がユーザの入力によって変化するようにする. これにより, マウスの動きや, メニューアクションに対する処理を図形文法の一部として記述することができる.

マウスを扱う終端記号は, 位置を示す座標(pos)とマウスボタンが押されているかどうかを示すフラグ(button[1-3])を属性として持つ. また, クリックやドラッグなどのアクションイベントを保持するための属性(action)と, また, その処理を行ったかどうかを示すフラグ(handled)を持つ. handled 属性は, イベントが発生したときに false に設定される. 文法でこのイベントを処理した際に, この属性を true に設定することにより, イベントが複数にバインドされることを防ぐことができる.

```
Mouse(pos, button1, button2, button3,
    action, handled)
```

ボタンを扱う終端記号は, ボタンの状態を示す属性(action)と, ボタンが押された際の処理を行ったかどうかを示すフラグ(handled)を持つ. この終端記号は, 図形言語毎に必要に応じて利用する. また, ボタン毎に生成され, これを区別するために, 終端記号名に名前が付加される. なお, action 属性には, ボタンが押されたときには“pressed”が設定される.

```
Button_name(current, handled)
```

メニューを扱う終端記号は, 選択されたメニュー項目

を示す属性(current)と，メニューが選択された際の処理を行ったかどうかを示すフラグ(handled)を持つ．この記号は，図形言語毎に必要に応じて利用する．また，メニュー毎に生成され，これを区別するために，終端記号名に名前が付加される．

```
Menu_name(current, handled)
```

チェックボタンを扱う終端記号は，現在選択されているボタンを示す属性(current)と，状態が変化した際の処理を行ったかどうかを示すフラグ(handled)を持つ．この記号は，図形言語毎に必要に応じて利用する．また，チェックボタンのグループ毎に生成され，これを区別するために，終端記号名に名前が付加される．

```
CheckButton_name(current, handled)
```

3 図形文法による図形編集機能の記述例

提案した図形文法を用いて，図形編集機能を記述する例を示す．

3.1 図形の描画

チェックボタンで四角や線，矢印などを選択できるようになっていて，その状態に応じて，キャンバス上をドラッグすると指定された図形を描画するといった機能を文法で記述する．

```
D:DrawRect ::= C:CheckButton_Mode,
    M:Mouse where (
  C.current == "rectangle" &&
  M.action == "button1-press" &&
  M.handled == false
) { } {
  M.handled = true
  N = createObject("rectangle")
  N.start = M.pos
  N.end = M.pos
  N.selected = true
} { }

D:DrawRectDrag ::= C:CheckButton_Mode,
    M:Mouse, O:Object where (
  C.current == "rectangle" &&
  M.button1 == "pressed" &&
  O.selected == true
) { } {
  D.ct = addConstraint("M.pos == O.end")
}
  removeConstraint(D.ct)
  O.selected = false
}
```

生成規則中の *createObject*() はキャンバス上に引数で指定された図形を追加するとともに，新たに終端記号 Object を生成する関数であり，新しい終端記号を示す識別子を返す．生成された終端記号 Object はキャンバスの図形と対応しており，その属性が変更されると，キャンバス上の図形も変更されるようにバインディングされている．*addConstraint*() は属性間に制約を課す関数であり，引数で指定した制約を課すことができる．また，この関数は，その制約を示す識別子を返す．課せられた制約は *removeConstraint*() で外される．指定された制約は，制約解消系に設定され，ある属性値が変更された場合に，指定された制約を満たすように属性値が変更される．

非終端記号 DrawRect の生成規則は，マウスのボタンが押されたときに，図形を作成するためのもので，Mouse の action 属性でボタンが押されたことを判定している．また，認識された後は *Construct_script* 中で handled 属性を false に設定し，このイベントに対するバインドが行われたことを示す．この生成規則は，*Construct_script* 中で handled 属性が変化するため制約が満たされなくなり，非終端記号 DrawRect は直ちに削除される．

非終端記号 DrawRectDrag の生成規則は，ドラッグ中の処理を記述したものである．非終端記号 DrawRect の生成規則が適用された際に作成された図形を，四角形の頂点とマウスの位置を等しくする制約を用いて変形させている．マウスのボタンが離された場合は，Mouse の button1 属性が“pressed”なくなるため DrawRectDrag は削除され，*Finalize_script* に記述されている，制約の解除が実行される．

この例では，四角形を描画するが，同様の記述をすることにより，線や矢印などを描画する生成規則を記述できる．

3.2 図形の移動

図形の上でマウスのボタン1をドラッグすると，その図形がマウスに追従して移動する例を示す．

```
M:ObjectMove ::= M:Mouse, O:Object where
    (
  M.button1 == "pressed" &&
  isOverlapped(O, M.pos)
```

```
) { } {
  M.ct = addConstraint("O.mid == M.pos")
} {
  removeConstraint(M.ct)
}
```

isOverlapped()は, 図形とマウスの座標が重なっているかどうかを判定する関数である. マウスが図形上で押されている間, この生成規則が適用される. マウスが押された時にマウスの位置と図形の位置を等しくなるように制約解消系に設定することによって, 図形がマウスとともに動くようになる. マウスが放された時にはこの制約を外すようになっている.

3.3 図形の選択

チェックボタンで図形選択モードになっている場合に, 図形上でクリックすることで図形の選択を行う例を示す.

```
S:ObjectSelect ::= C:CheckButton_Mode,
    M:Mouse, O:Object where (
  C.current == "select" &&
  M.action == "button1-click" &&
  M.handled == false &&
  isOverlapped(O, M.pos)
) { } {
  M.handled = true
  O.selected = !O.selected
} { }
```

図形の上でクリックされると, この生成規則が適用され, 終端記号 Object の selected 属性を反転する. この規則は, *Construct_script* 中で handled 属性を変更しているため, 非終端記号 ObjectSelect は直に削除される. なお, 終端記号 Object は, selected 属性が true の場合, 図形にハンドルを付加して選択状態を示すようにバインディングがなされている.

3.4 ボタンのバインド

ボタンが押された場合の処理について例を示す. ここでは, 図形が選択状態にあるもの(終端記号Objectのうち selected 属性が true であるもの)を削除するものとする.

```
E:ObjectDelete ::= B:Button_Delete
    O:Object where (
  B.action == "pressed" &&
  B.handled == false &&
  O.selected == true
) { } {
  deleteObject(O)
} { }

E:ObjectDeleteFinish ::=
    B:Button_Delete where (
  B.action == "pressed" &&
  B.handled == false &&
  not_exist O:Object where (
    O.selected == true
  )
) { } {
  B.handled = true
} { }
```

非終端記号 ObjectDelete の生成規則は, 選択状態にある図形を削除している. なお, *deleteObject*()は引数に指定された終端記号を削除する関数である.

非終端記号 ObjectDeleteFinish の生成規則は, 削除する図形が無くなった場合の処理を記述している. イベントバインドが終了した事を示すため handled 属性を変更している. なお, 選択された図形がないという制約条件は not_exist を用いて記述している.

4 議論

Construct_script と *Finalize_script* に加え, 非終端記号が存在している間は, 制約解消系を用いて属性間の関係を指定することにより, ユーザの入力に対する一連の処理について図形文法を用いて簡潔に記述できるようになる. また, 認識された図形の意味に応じたイベントのバインディングも容易になる. また, エディタと解析部が統合されており, インタラクティブに処理が行われるため, 図形の状態に応じて, システムに変更が可能であり, 解析結果に応じて図形文法に変更を加えるといった, リフレクティブなシステムが実現することができる.

Eviss では, *Construct_script* と同様の機能を提供する Action を導入しているが, 今回我々が提案した手法では, LHS の記号が削除された場合の処理が記述できるため, 後処理が簡単に記述できる.

5 まとめ

図形言語処理システムにおける, 図形エディタと空間解析器の統合について述べた. 2つを統合するために, 図形文法を拡張し, さらに, 図形編集機能を図形文法で記述した. これにより, インタラクティブな図形言語

処理システムが図形文法を用いて記述できるようになった. 今後は, 具体的な図形言語を用いて, 本手法の有効性を確認したいと考えている.

参考文献

[1] G. Costagliola, A. D. Lucia, S. Orefice, and G. Tortora, “Positional Grammars: a Formalism for LR-like Parsing of Visual Languages”, Visual Languages Theory, Springer, pp.171-191, 1998.

[2] F. Ferrucci, G. Tortora, M. Tucci, and G. Vitiello, “A Predictive Parser for Visual Languages Specified by Relation Grammars”, Proceedings of IEEE Symposium on Visual Languages, pp.245-252, 1994.

[3] R. Helm, K. Marriott, and M. Odersky, “Building Visual Language Parsers”, Conference proceedings on Human Factors in Computing Systems (CHI’91), pp.105-112, 1991.

[4] E. J. Golin and T. Maglierry, “A Compiler Generator for Visual Languages”, Proceedings of IEEE Symposium on Visual Languages, pp.314-321, 1993.

[5] L. R. Balt, “Full CMG parsing”, Master's thesis, Leiden University, The Netherlands, 1996.

[6] 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.

[7] 藤山 健一郎, 田中 二郎, “例示入力図を用いた Spatial Parser Generator”, 日本ソフトウェア科学会第 17 回大会, CD-ROM 予稿集, 4pages, 2000.

[8] S. S. Chok and K. Marriott, “Automatic Construction of Intelligent Diagram Editors”, Proceedings of the ACM Symposium on User Interface Software and Technology, pp.185-194, 1998.