代数的仕様記述を利用したプログラムのエラー記述変更

3S－3

濱口毅　山本晋一郎　阿草清滋
名古屋大学工学部

1　はじめに

著者らはこれまで，代数的仕様記述におけるエラー記述について研究を行ってきており，これまでエラー処理に関する記述のない仕様に対してエラー記述を自動付加する方法を提案してきた．本論文ではこれを応用し，プログラムに対してエラー処理のための記述を付加する方法について考察する．

代数的仕様は形式性に優れ，データ構造の仕様決定，及びその検証，解析を数学的な手法を用いて行うことができる．一方，ソフトウェア開発においてエラー処理は不可欠であり，ソフトウェアの機能拡張に際して，エラー処理に関する記述の変更が必要となることが多い．実用的なソフトウェアにおいては，正常処理よりもエラー処理に関する記述が多くなり，この変更がプログラムに与える影響が大きいため，これを誤りなく自動化する方法が強く望まれる．

本稿ではソフトウェアのエラー処理に関する記述の変更を，代数的仕様のレベルで行い，C言語で記述されたプログラムに反映させる方法を探る．これにより形式性の確保と記述性の向上の両立をはかることを目指す．

2　代数的仕様とプログラムの対応

ソフトウェアのエラー処理に関する記述の変更を，代数的仕様のレベルで行い，これをプログラムに反映させるためには，変更の対象となる仕様とプログラムがある程度対応が取れていなくてはならない．

形式的仕様の実現法には，仕様を実行可能なものとみなし実行する方法とプログラマが仕様上の決定を行いながら，プログラムを作成する方法と2種類ある．しかし，実行可能な仕様を書くことは，仕様記述の大きな制限となる．また，代数的仕様だけで作成しようとするシステムの仕様をすべて記述することは現実的ではない．本稿では仕様は実行可能でないものとみなし，プログラムは人手により作成されるという立場をとる．このアプローチに基き，仕様とプログラムの対応をとるときに問題となる，データの表現の対応について以下で述べる．

プログラマは，抽象データ型の記述された仕様を基に具体的に使用するデータ型を選択しながら，プログラムを作成していく．仕様とデータ型の対応をとるための確実な手段は，プログラムを作成する際に，仕様とプログラムの対応を記録していくことである．例えばこの対応をプログラム中にコメントとして埋め込んでおけば，データ型の対応は明らかになる．

例えば自然数のリストの仕様では，構成子は次のような二つの関数である．

$$\mathrm{nil}: \rightarrow \mathrm{List}$$
$$\mathrm{cons}: \mathrm{Nat}, \mathrm{List} \rightarrow \mathrm{List}$$

リストの仕様をC言語により実現したプログラムでは，リストは構造体で表現される場合もあれば，配列によって表現される場合もある．自然数のリストをC言語の構造体を使って実現すると例えばそのデータは以下のようになる．

```
structure list {
    Nat data;
    structure list *next;
}
```

この構造体は *cons* に対応するものであり，定数関数 *nil* に対応するのは `(struct list *) NULL` である．

仕様に書かれた抽象データ型を実現する時に，具象データ型の選択は実現の大きなポイントである．これらのデータ型の対応を自動的に見つけ出すことは一般的に難しい．したがって，データ型を決定する際にこの対応をプログラマが記録することが必要である．そして，この情報をコメントしてプログラムに埋め込み，仕様変更のプログラムへの反映に利用する．

3　エラー記述の付加とプログラム変更の例

図1.のような自然数のリストと，リストから要素を削除する関数deleteの仕様について考える．

```
sort List
  constructor
```

Modification of Error Description using Algebraic Specification

Takeshi Hamaguchi, Shinichirou Yamamoto, Kiyoshi Agusa
Nagoya University
Furo, Chikusa, Nagoya 464-01, Japan

```
    nil: → List
    cons: Nat, List → List
  defined
    delete: Nat, List → List
  equation
    delete(y, cons(x, list))
      == if(eq(x, y), list, cons(x, delete(y, list)))
  end
```

図 1. 自然数のリストの仕様

図 1. の仕様を実現すると例えば図 2. のようなプログラムになる.

```
1: struct list *delete(Nat n, stluct list *l)
2: {
3:     if (l->data == d) {
4:         return (l->next);
5:     } else {
6:         l->next = delete(d, l->next);
7:         return l;
8:     }
9: }
```

図 2. 自然数のリストのプログラム

図 2. のプログラムでは, リストの delete のコードでは, その第 2 引数が NULL でないことを仮定している.

図 1. の仕様では, delete の第 2 引数に *nil* が来たときの等式が記述されておらず, もし *nil* が来たときはエラーとなる. そこで [1] のアルゴリズムにより, 次のような関数, エラー構成子と等式を追加する.

```
  constructor
    err_list: → List
  equation
    delete(n, nil) == err_list
```

この等式は, エラー値を発生させるためのものであり, このような等式をエラー生成等式と呼ぶ.

C 言語のプログラムにおいて, エラーは関数の返り値として特別な値を用意して表現されることが多い. エラー構成子はまさにこの特別な値に対応する. エラー生成等式はプログラムでは以下のようなコードになる.

```
if (l == (structure list *) 0)
    return ERR_LIST;
```

このコードを, delete 関数を実現したプログラム中に挿入すればよいが, 一般にはエラー処理のための記述を挿入する場所を特定するのは簡単ではない. そこで抽象実行を利用し, 仕様の等式に対応するプログラムコードを挿入する場所を見つける必要がある. エラー生成等式から, エラー発生の条件は, delete の第 2 引数のリストに依存することがわかる. 抽象実行により, プログラムの 3–7 行めは, 図 1. の仕様の等式を実現した部分であり, エラーが発生した時にこの部分が実行されないようにエラー生成等式に対応するコードをプログラムに挿入する必要がある. さらに, 依存解析することにより, エラー処理のためのコードを挿入すべき位置を特定できる. この場合は, 3 行めの直前に挿入すればよいことがわかる.

この例では実現されてできたプログラムが単純であるため, このコードは delete 関数の最初の部分に挿入すればよいことがすぐわかるが, 一般には以上で述べた方法を用いる必要がある.

また, [1] では, エラー構成子を仕様に追加しても, 矛盾が起きないように, エラーを伝播させるための等式, エラー伝播式を導入している. エラー伝播式を実現したコードもプログラムに挿入する必要があるが, ここでは割愛する.

4 おわりに

ソフトウェアのエラー処理に関する記述の変更を, 代数的仕様のレベルで行い, C 言語で記述されたプログラムに反映させる方法について述べた. これによりプログラム作成のコストを大きく減らすことが期待できる. 仕様の変更をプログラムへ反映させるために, プログラムを変更を箇所を見つけ出す必要があるが, これに依存解析や抽象実行が利用できる. しかし, エラー記述の自動付加に利用できるような抽象実行システムは今のところない. 著者らの研究室で開発された CASE ツールプラットフォーム Sapid[2] を用いてこのようなツールを作成することが可能であると考えられる. さらに, これらの技術を用いてエラー処理に関する記述の変更を行うことが今後の課題である.

参考文献

[1] 濱口毅, 酒井正彦, 山本晋一郎, 阿草清滋. エラーつき代数的仕様とエラー記述の自動付加. 電子情報通信学会 論文誌, Vol. J78-D-I, No. 3, pp. 323–330, 1995.

[2] 山本晋一郎, 阿草清滋. 細粒度リポジトリに基づいたツール・プラットフォームとその応用. 情報処理学会 研究報告, 第 95-SE-102 巻, pp. 37–42, 1995.