ソフトウェア工学 73－6
(1990. 7. 19)

代数的仕様への変換による
前提終了条件表記仕様のデバッグについて

小野 康一* 山本 剛** 所 洋一*** 深澤 良彰* 門倉 敏夫*

* 早稲田大学 理工学部 ** ソニー㈱ CAE推進本部 *** NHK 富山放送局

前提終了条件表記の仕様を代数的仕様へ変換して提示する．これにより，記述性のよい前提終了条件表記の仕様で記述し，理解性のよい代数的仕様でそのデバッグをすることが可能となる．本変換手法では，まず，抽象データ型に対する2つの操作を組み合わせて記号実行し，結果状態が同じものを組み合わせて等式とする．その等式から5つの選択規準を用いて公理を選択する．

この論文では，仕様中にエラーを埋め込み，本変換手法により変換された公理中にそのエラーが反映しているかどうかを調べ，本変換手法の利点と欠点を述べる．

Specification Debugging
by Transforming Representation

Kouichi ONO*, Tsuyoshi YAMAMOTO**, Youichi TOKORO***, Yoshiaki FUKAZAWA* and Toshio KADOKURA*

* School of Science and Engineering, Waseda University, 3-4-1, Okubo, Shinjuku-ku, Tokyo 169, Japan.
** Sony Corporation, 6-7-35, Kitashinagawa, Shinagawa-ku, Tokyo 141, Japan.
*** NHK Toyama Broadcasting Station, 3-1, Shinsougawa, Toyama-city 930, Japan.

It is important to debug a large scale specification. For this purpose, we have developed a transformation method from a specification in pre-post conditional notation, which is easy to write, into an axiomatic specification, which is easy to understand. Roughly speaking, this method consists of the following steps; in the first step, all possible operational compositions for a given abstract data type are symbolically executed. Two compositions which result the same status are combined into an equation. Lastly, a reasonable set of equations are selected by our heuristic selection rules.

In this paper, we illustrate the transformation process of an erroneous example and the effectivity of this method.

１．はじめに

ソフトウェア開発過程に対する形式的仕様の導入は，求められるソフトウェアの正確な定義を与えるのに重要な役割を果たす[1]．

信頼性の高いソフトウェアを作成するための一手法として，抽象データ型を用いてソフトウェアを定義することが多く研究されている[2]．抽象データ型は，モジュール化・抽象化・情報隠蔽・局所化など，プログラミング方法論における重要な概念を含んでいる．この抽象データ型を厳密かつ形式的に定義する方法として，代数的仕様記述法が提案されている[3]． また，抽象データ型の操作仕様などを示すために前提条件と終了条件を論理式で表記する記述法も提案されている[4]．

前提終了条件表記の仕様記述法では，抽象データ型に対する操作を行なう前に成立すべき条件（状態）と操作後に成立すべき条件（状態）を記述する．この仕様記述法は操作の前後における状態のみの記述であるため，比較的記述性がよい．しかし，記述が操作的になりがちで，理解性に欠くことがある．

代数的仕様記述法では，抽象データ型を代数とみなし，代数の計算系を公理と呼ばれる等式の集合で定義する．公理は直感的ではあるが，計算系の規模が大きくなるにつれて，それを完全に定義する公理の集合を与えるのはかなりの熟練を要する．

本研究は，抽象データ型の操作を前提終了条件で記述した仕様から，操作の間に成立する等価関係を生成するための手法を与える[5]． 本研究は，この等価関係を公理とする代数的仕様を記述者に提示することにより，仕様のデバッグを支援することを目的としている．

本手法による変換は大きく分けて，操作の等価関係の生成と，生成した等式の選択，の2つの処理手順からなる．等式の生成は，前提終了条件表記の仕様を基に，経験的に得られた基準を用いて表現形式を変換することによって行なう．等式の選択は，経験的に得た選択規準を用いて，生成した等式群に制限を加え，公理として出力する等式の数を適当な量に抑制することによって行なう．

２．本手法の概要

本手法の入力として，前提終了条件表記の仕様記述言語を使用する．また出力は，入力の仕様に操作間の公理を付加した形式である．公理の表記には，代数的仕様の形式を用いる．

以下では，本手法の入出力の概略について説明する．なお，本手法の入力となる前提終了条件表記の仕様記述言語による定義例を付録に与える．ただし，この例においては，操作モジュールｔｏｐの定義に誤りを与えてある．

２．１　操作モジュール仕様

本手法の入力は，操作モジュールを前提終了条件表記で定義した仕様である（図１）．操作モジュールの仕様は，インタフェース部，モジュール仕様記述部，及び，型／オブジェクト宣言部からなる．更に，モジュール仕様の階層的記述も可能である．

インタフェース部では，抽象データ型の各操作モジュールに対するパラメータの宣言や，他の操作モジュールで定義された操作やデータ型等を利用するためのＩＭＰＯＲＴ宣言，他の操作モジュール等で利用可能とするためのＥＸＰＯＲＴ宣言等を行なう．

モジュール仕様記述部では，前提終了条件表記により，抽象データ型の操作仕様を記述する．

型／オブジェクト宣言部では，モジュール仕様内部で使用するデータ型の宣言や，データオブジェクトの宣言を行なう．

２．２　操作間の公理

本手法の出力は，操作モジュールの間に成立する公理を，入力の仕様に付加した形式である．操作間の公理（図２）は，等式の集合で与える．

図1．操作モジュール仕様

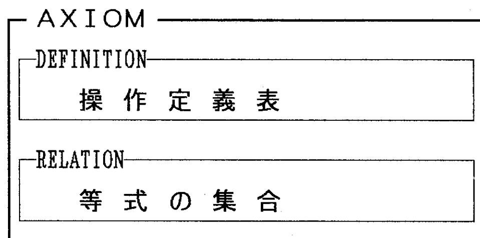

図2．操作間の公理

3．仕様変換システム

本手法に基づく仕様表現形式変換システムを開発中である．以下でその詳細を説明する．

3．1　システムの構成

本システムの構成を図3に示す．

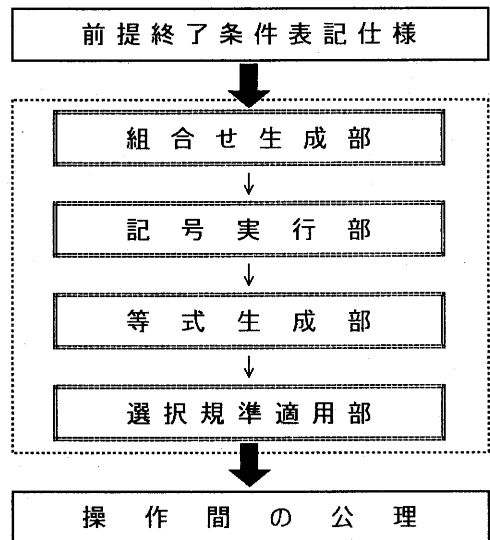

図3．システム構成図

3．2　仕様変換手法

前提終了条件表記仕様を代数的仕様に変換する場合，まず，各操作モジュールのパラメータに，そのパラメータのデータ型と同じデータ型を出力する操作モジュール，定数，変数を組み合わせて，二つの操作の合成操作を作成する．そして，合成操作の評価結果が同じ状態のものを結合した等式を生成すれば，公理の候補を生成することが可能である．厳密には，十分な公理の集合を得るために必要な合成が，二つの操作の組合せで実現できるとは言えない．しかし，二つで十分である場合が多いことが経験から得られている．

等式を，二つの操作の合成から生成した場合，等式の数は，抽象データ型における操作数が増加するにともなって，操作数の二乗に比例して増加する．そこで，以下で述べるような手順により，生成した等式から，意味のある適当な量の等式を公理として選択するための変換手法を考案した．この手法は，公理選択のための規準を用いて，生成した等式の数を削減する．この公理選択のための規準を，選択規準と呼ぶことにする．

しかし，この手法では，前提終了条件表記仕様で記述された内容全体を，代数的仕様に反映している

訳ではないため，エラーの発見が不可能な場合もある．このような場合には，選択規準による選択のレベルを緩和することにより，エラーを公理まで波及させるようにしている．

以下では，抽象データ型スタックの仕様例を用いて，本システムの各構成要素を説明する．仕様の全記述は付録に与える．なお，操作モジュール仕様の一箇所に，誤った定義を与えてある．

3．2．1　組合せ生成部

組合せ生成部は，操作モジュールの入出力パラメータのデータ型をもとに，組合せを行なう操作を選択し，合成操作を生成する．組合せを行なう場合，ある抽象データ型の操作の入力パラメータとして，そのパラメータのデータ型と同じデータ型の変数・定数，あるいは，そのデータ型と同じデータ型の値を返す操作を用いる．ここで，ある操作の入力パラメータに用いる操作には，それと同じ操作は重複して使用しないものとする．また，ある操作の入力パラメータとして用いた他の操作自身のパラメータには，変数のみの使用を許すものとする．この規準は，われわれの経験に基づいて定めた．

付録に示した抽象データ型スタックの仕様を本手法で変換した時の，組合せ生成部の出力結果を図4に示す．組合せによる合成操作および状態は22通りであった．なお，【】内の最初の数字は，組合せに取り出した操作・状態のグループを表す．

```
[1-1]   creates()

[2-1]   push( s1, e1 )
[2-2]   push( s1, top(s2) )
[2-3]   push( creates(), e1 )
[2-4]   push( pop(s1), e1 )

[3-1]   pop( s1 )
[3-2]   pop( creates() )
[3-3]   pop( push(s1,e1) )

[4-1]   top( s1)
[4-2]   top( creates() )
[4-3]   top( push( s1, e1 ) )
[4-4]   top( pop( s1 ) )

[5-1]   emptys( s1 )
[5-2]   emptys( creates() )
[5-3]   emptys( push( s1, e1 ) )
[5-4]   emptys( pop( s1 ) )

[6-1]   s1
[6-2]   s2
[6-3]   e1
[6-4]   TRUE
[6-5]   FALSE
```

図4．組合せ生成部の出力

3．2．2　記号実行部

記号実行部は，組合せ生成部が生成した操作の組合せについて，その出力のパラメータ変数の値等を計算する．

図4の組合せを入力とした時の，記号実行部の出力の一部を図5に示す．なお，図中の**E**は，操作の前提条件が成立せずに状態遷移しなかった場合を示す．

```
[1-1]   creates() :=
            {s.s=[],
             s.p=0 }

[2-1]   push(s1,e1) :=
            {s.s=[S1[0],…,S1[s1.p-1],e1],
             s.p=s1.p+1}

[2-2]   push(s1,top(s2)) :=
            IF emptys(s2)=TRUE :
                {s.s=[S1[0],…,S1[s1.p-1]],
                 s.p=s1.p},                              E
            IF emptys(s2)=FALSE :
                {s.s=[S1[0],…,S1[s1.p-1],
                     S2[s2.p]],
                 s.p=s1.p+1}

[2-3]   push(creates(),e1) :=
            {s.s=[e1],
             s.p=1}

[2-4]   push(pop(s1),e1) :=
            IF emptys(s1)=TRUE :
                {s.s=[e1],
                 s.p=1},                                 E
            IF emptys(s1)=FALSE :
                {s.s=[S1[0],…,S1[s1.p-2],e1],
                 s.p=s1.p}

[3-1]   pop(s1) :=
            IF emptys(s1)=TRUE :
                {s.s=[],
                 s.p=0},                                 E
            IF emptys(s1)=FALSE :
                {s.s=[S1[0],…,S1[s1.p-2]],
                 s.p=s1.p-1}

[3-2]   pop(creates()) :=
            {s.s=[],
             s.p=0}                                      E

[3-3]   pop(push(s1,e1)) :=
            {s.s=[S1[0],…,S1[s1.p-1]],
             s.p=s1.p}
                    ⋮
```

図5．記号実行部の出力

３．２．３　等式生成部

等式生成部は，記号実行部が計算した出力のパラメータ変数の値を，全ての組合せについて比較し，値が全て等しい組合せを選んで等式を作成する．この時，操作モジュールに条件分岐が含まれる場合は，各条件毎に組合せ，等式を作成する．

図５の組合せを入力とした時の，等式生成部の出力の一部を図６に示す．全ての組合せに対して生成された等式の数は，28であった．

```
《s.s=[], s.p=0》
[1-1]    creates()
  =[3-1]IF emptys(s1)=TRUE : pop(s1)
[1-1]    creates()
  =[3-2]pop(creates())
[3-1]    IF emptys(s1)=TRUE : pop(s1)
  =[1-1]creates()
[3-1]    IF emptys(s1)=TRUE : pop(s1)
  =[3-2]pop(creates())
[3-2]    pop(creates())
  =[1-1]creates()
[3-2]    pop(creates())
  =[3-1]IF emptys(s1)=TRUE : pop(s1)
《s.s=[e1], s.p=1》
[2-3]    push(creates(),e1)
  =[2-4]IF emptys(s1)=TRUE:push(pop(s1),e1)
[2-4]    IF emptys(s1)=TRUE : push(pop(s1),e1)
  =[2-3]push(creates(),e1)
《s1:={s.s=[S1[0],…,S1[s1.p-1]], s.p=s1.p}》
[3-3]    pop(push(s1,e1))
  =[6-1]s1
              ⋮
```

図６．等式生成部の出力

３．２．４　選択規準適用部

選択規準適用部は，等式生成部が生成した各等式について，５種類の選択規準でふるいにかける．選択基準の適用は規準①から⑤の順としている．

３．２．４．１　選択規準①

等式の右辺における操作の入力パラメータ中に現われる変数は，必ず左辺にもなければならない．即ち，左辺中に存在していない変数が，右辺に現われている等式は，ふるい落とす．

等式生成部で生成した等式に対して，選択規準①を適用することにより，図７に示す等式をふるい落とした．

```
[1-1]creates()
  =[3-1]IF emptys(s1)=TRUE : pop(s1)
[3-2]pop(creates())
  =[3-1]IF emptys(s1)=TRUE : pop(s1)
[2-3]push(creates(),e1)
  =[2-4]IF emptys(s1)=TRUE : push(pop(s1),e1)
[5-2]emptys(creates())
  =[5-1]IF emptys(s1)=TRUE : emptys(s1)
[5-2]emptys(creates())
  =[5-4]IF emptys(pop(s1))=TRUE :
                              emptys(pop(s1))
[5-1]IF emptys(s1)=FALSE : emptys(s1)
  =[5-3]emptys(push(s1,e1))
```

図７．選択規準①の適用

３．２．４．２　選択規準②

等式の右辺における操作に対応するモジュール仕様内の分岐条件は，左辺の操作モジュール自身に依存してはいけない．即ち，右辺中の操作モジュールにおける分岐条件文で，左辺の操作モジュールを呼出しているような等式は，ふるい落とす．

選択規準①を適用して残った等式に対して，選択規準②を適用することにより，図８に示す等式をふるい落とした．

```
[5-1]IF emptys(s1)=TRUE : emptys(s1)
  =[5-4]IF emptys(pop(s1))=TRUE :
                              emptys(pop(s1))
[5-4]IF emptys(pop(s1))=TRUE: emptys(pop(s1))
  =[5-1]IF emptys(s1)=TRUE : emptys(s1)
[5-1]IF emptys(s1)=FALSE : emptys(s1)
  =[5-4]IF emptys(pop(s1))=FALSE :
                              emptys(pop(s1))
[5-3]emptys(push(s1,e1))
  =[5-1]IF emptys(s1)=FALSE: emptys(s1)
[5-3]emptys(push(s1,e1))
  =[5-4]IF emptys(pop(s1))=FALSE :
                              emptys(pop(s1))
[5-1]IF emptys(s1)=FALSE : emptys(s1)
  =[5-4]IF emptys(pop(s1))=FALSE :
                              emptys(pop(s1))
[5-3]emptys(push(s1,e1))
  =[5-4]IF emptys(pop(s1))=FALSE :
                              emptys(pop(s1))
```

図８．選択規準②の適用

３．２．４．３　選択規準③

等式の左辺における操作の数は，右辺の操作数以上でなければならない．即ち，右辺の操作数以上の操作が，左辺にあるような等式は，ふるい落とす．

選択規準②を適用して残った等式に対して，選択規準③を適用することにより，図９に示す等式をふるい落とした．

```
[1-1]creates() = [3-2]pop(creates())
[3-1]IF emptys(s1)=TRUE : pop(s1)
              = [3-2]pop(creates())
[5-1]IF emptys(s1)=TRUE : emptys(s1)
              = [5-2]emptys(creates())
```

図９．選択規準③の適用

３．２．４．４　選択規準④

等式の左辺における操作の組合せが同じで，右辺が異なっている等式が複数個存在する場合には，次の規準を順に適用し，選択する．

ａ）等式の左辺の操作数と右辺の操作数の差が大きい方を選択する．

ｂ）等式の左辺の変数の数と右辺の変数の数が大きい方を選択する．

選択規準③を適用して残った等式に対して，選択規準④を適用することにより，図１０に示す等式をふるい落とした．

```
[5-4]IF emptys(pop(s1))=TRUE : emptys(pop(s1))
              = [5-2]emptys(creates())
```

図１０．選択規準④の適用

３．２．４．５　選択規準⑤

等式中の操作に対応するモジュール仕様内に条件分岐が含まれている場合は，他の組合せ中に，全ての条件が揃っていなければならない．即ち，すでに他の条件の等式が，選択基準によって落とされた等式は，ふるい落とす．

選択規準④を適用して残った等式に対して，選択規準⑤を適用することにより，図１１に示す等式をふるい落とした．

```
[3-1]IF emptys(s1)=TRUE : pop(s1)
    = [1-1]creates()
[2-4]IF emptys(s1)=TRUE : push(pop(s1),e1)
    = [2-3]push(creates(),e1)
[5-1]IF emptys(s1)=TRUE : emptys(s1)
    = [6-4]TRUE
[5-4]IF emptys(pop(s1))=TRUE : emptys(pop(s1))
    = [6-4]TRUE
[5-1]IF emptys(s1)=FALSE : emptys(s1)
    = [6-5]FALSE
[5-4]IF emptys(pop(s1))=FALSE: emptys(pop(s1))
    = [6-5]FALSE
```

図１１．選択規準⑤の適用

３．３　誤りを含んだ仕様の変換例

スタックの仕様に対して本手法を適用し，等式の生成，選択基準の適用を行なうと，図１２のような等式の集合が公理として残る．

```
[3-2]pop( creates() ) = [1-1]creates()       (1)
[3-3]pop( push(s1,e1) ) = [6-1]s1            (2)
[4-3]top( push(s1,e1) ) = [6-3]NULL          (3)
[5-2]emptys( creates() ) = [6-4]TRUE         (4)
[5-3]emptys( push(s1,e1) ) = [6-5]FALSE      (5)
```

図１２．公理

付録に示したスタックのモジュール仕様は，操作モジュールｔｏｐの終了条件部に誤りを含んでいる．図１２第(3)式を見ると，スタックs1に要素e1をプッシュした後の，スタックのトップの要素がNULLになっているのがわかる．しかし，正しくはスタックのトップの要素はe1となっていなければならないので，topかpushのどちらかの操作モジュールが意図するものと違っているのがわかる．この判定は，仕様記述者自身が行なう．

４．評価

本手法の評価として，４つの抽象データ型の操作仕様について本手法を適用し，その有効性を考察した．それぞれの抽象データ型の操作数，生成された等式数，及び，選択規準を適用した回数を表１に示す．このうち，STACK，QUEUE，LISTの操作仕様については，出力として得られた等式にエラーが反映した．しかし，EDITORの操作仕様については選択規準⑤を適用すると等式が残らないため，選択基準の適用判断をゆるめ，選択基準④までを適用している．

表１．各ＡＤＴのサイズと選択規準適用回数

ＡＤＴ名		STACK	QUEUE	LIST	EDITOR
操作数		５	５	６	９
選択規準	①	６	８	４	１５６
	②	７	４	４	２２
	③	３	２	０	７５
	④	１	１	０	０
	⑤	６	５	０	—
	計	２３	２０	８	２５３
等式数		５	５	２	１４

本手法は発見的な方法であるため，エラーの種類によって，変換後の等式にそのエラーが反映しなかったり，複数のエラーによって等式にエラーが反映しないこともある．前者の場合は，選択規準を⑤から順にゆるめることによって対応可能である．後者の場合は，本手法では発見不可能である．

５．おわりに

本研究では，前提終了条件表記により記述された仕様から代数的仕様に変換する手法を与えた．本仕様変換手法では，まず，抽象データ型に対する２つの操作を組み合わせて，それを記号実行し，結果状態が同じものを組み合わせて等式とする．等式の組合せを削減するために，その等式に対して，５つの選択規準を用いて制限を加え，公理を選択している．この手法は，記述性のよい前提終了条件表記の仕様から，動作が直感的でわかりやすい代数的仕様に，仕様の表現形式を変換することにより，前提終了条件表記の仕様に含まれるエラーの検出を支援する．

選択規準は，あくまで経験的なものであるため，適用によっては，エラーの含まれている等式が残らないこともある．しかし，ある操作仕様中のエラーは，等式生成時に組合せによって多数の等式に波及するので，その内の一つでも残れば，デバッグ支援のための情報としては充分である．

本システムは，仕様記述者が前提終了条件表記の仕様のレビューを行なう際に利用するためのツールとして開発された．前提終了条件表記の仕様の完全性を確認するには，他のテスト手法を採用すべきである．

本論文では，標準的な選択基準として，５つの基準を提案した．しかし，仕様の記述法は適用するアプリケーションの種類や記述者により，大きく異なる．したがって，より正確なシステムに近づけるためには，このような条件を選択基準に反映させる必要がある．このために，アプリケーションの種類などに基づく条件をエキスパートシステム化し，手法の改良を行なっていく予定である．

参考文献

[1] Cohen, B., Harwood, W.T. and Jackson, M.I., "The Specification of Complex Systems", Addison-Wesley Publishing Company, 1986.

[2] Belkhouche, B. and Urban, J.E., "Direct Implementation of Abstract Data Types from Abstract Specification", IEEE Trans. on Software Eng., Vol. SE-12, No. 5, pp. 649-661, 1986.

[3] 稲垣康善，"抽象データ型の概念と仕様記述法"，情報処理，Vol. 27, No. 2, pp. 120-128, 1986.

[4] Beichter, F., Herzog, O. and Petzsch, H., "SLAN-4 — A Software Specification and Design Language", IEEE Trans. on Software Eng., Vol. SE-10, pp. 155-162, 1984.

[5] 山本 剛，所 洋一，小野康一，深澤良彰，門倉敏夫，"表現形式の変換による仕様のデバッグ"，情報処理学会第40回全国大会，2R-8, pp. 1031-1032, 1990.

付録．スタックの操作モジュール仕様

```
MODULE creates() => stack
    INTERFACE
        s       : PARAMETER(WRITE)stack.
        stack   : IMPORT(TYPE).
    ENDINTERFACE
    SPECIFICATION
        PRE-creates  : TRUE
        POST-creates : s.p'=0
    ENDSPECIFICATION
ENDMODULE creates

MODULE push(stack,ElemType) => stack
    INTERFACE
        s       : PARAMETER(READ/WRITE)stack.
        e       : PARAMETER(READ)ElemType.
        stack   : IMPORT(TYPE).
    ENDINTERFACE
    SPECIFICATION
        PRE-push  : TRUE
        POST-push : s.s'[s.p]=e AND s.p'=s.p+1
    ENDSPECIFICATION
ENDMODULE push

MODULE pop(stack) => stack
    INTERFACE
        s       : PARAMETER(READ/WRITE)stack.
        stack   : IMPORT(TYPE).
    ENDINTERFACE
    SPECIFICATION
        PRE-pop  : emptys(s)=FALSE
        POST-pop : s.p'=s.p-1
    ENDSPECIFICATION
ENDMODULE pop

MODULE top(stack) => ElemType
    INTERFACE
        s       : PARAMETER(READ)stack.
        e       : PARAMETER(WRITE)ElemType.
        stack   : IMPORT(TYPE).
    ENDINTERFACE
    SPECIFICATION
        PRE-top  : emptys(s)=FALSE
        POST-top : e' = s.s[s.p]            ←誤り
    ENDSPECIFICATION
ENDMODULE top

MODULE emptys(stack) => BOOLEAN
    INTERFACE
        s       : PARAMETER(READ)stack.
        x       : PARAMETER(WRITE)BOOLEAN.
        stack   : IMPORT(TYPE).
    ENDINTERFACE
    SPECIFICATION
        PRE-emptys  : TRUE
        POST-emptys : IF s.p=0 : x' = TRUE
                      ELSE x' = FALSE
    ENDSPECIFICATION
ENDMODULE emptys
```