代数的言語で記述した抽象的順序機械型プログラムの設計検証の自動化

森岡澄夫[†] 北道淳司[†]
東野輝夫[†] 谷口健一[†]

本論文では，順序機械型プログラムの設計の正しさの形式的な証明を，半自動で行うための証明手順を提案する．証明を実際上可能とするために，要求仕様から始めて逐次，抽象的な一動作（遷移）をより具体的な動作の系列で展開（詳細化）していく階層的設計法を採用する．詳細化の度に，展開に用いた系列が元の遷移の動作を正しく実現していることを証明する．提案する証明手順は，制限された記述スタイルのもとで書かれた仕様に対し，構造的帰納法の各段階ごとに，その段階を証明するための式を不変表明などから作り，その式の成立を加減算と比較演算から成る整数上の論理式（プレスブルガー文）の恒真性判定手続きを用いて示す，というものである．従来は，項書き換えや場合分けなどの手法を検証者が複雑に組み合わせて証明していたため，証明に時間がかかり自動化も困難であった．しかし，提案する証明手順は単純なため自動化が容易であり，また，本論文で対象とするプレスブルガー文の恒真性は短時間で判定できるため高速である．提案する証明手順に基づく検証支援系を実際に作成した．その支援系は，検証者が基本述語・関数に関する補題のインスタンスや不変表明を与えれば，プレスブルガー文の合成やその真偽判定を自動で行う．本支援系を用いて，マックスソートプログラムの証明をCPU時間十数分程度で行えた．その証明では，式中の変数・演算子等の総出現数が1300以上の，大きな幾つかのプレスブルガー文の真偽を判定しなければならなかったが，いずれも数秒程度で判定できた．

Automatic Verification of Abstract Sequential Machine Style Programs Written in Algebraic Language

SUMIO MORIOKA,[†] JUNJI KITAMICHI,[†] TERUO HIGASHINO[†]
and KENICHI TANIGUCHI[†]

In this paper, we propose a proof method to prove semi-automatically the correctness of implementation of sequential machine style programs written in our algebraic language. We have restricted the description style of specifications and programs so that we can prove using the new proof method. In the restricted style, for example, to represent that a property holds for any element of an array, we use a predicate whose argument is the array. In our new proof method, we use a noetherian induction technique. We make some expressions which correspond to each induction step, and then determine the truth of each expression using a decision procedure for the prenex normal form Presburger sentences which are bounded only by universal quantifiers. The truth of such Presburger sentences can be determined quickly, even if they are rather large. At the proof using our verification support system, which is based on the new proof method, we have only to give the system some invariant assertions and some theorems for primitive functions/predicates. Using the system, we have carried out the proof of the correctness of a sorting program. The CPU time for the proof, including trial and error, was several hundreds seconds. The system has determined the truth of each Presburger sentence within a few seconds where the maximum length of the sentences exceeds 1300 symbols.

1. まえがき

プログラムが要求仕様を正しく実現していることを完全に保証するには，論理的な証明（検証）を行う必要がある．プログラムの要求性質が複雑な場合，それが正しく実現されていることの証明は複雑になる．自然語で証明をしていれば，証明が正しく行われていること，すなわち，証明の各ステップごと，そこで行う推論の前提条件が確かに成り立つことの確認は難しく

† 大阪大学基礎工学部情報工学科
Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University

なる．そのような場合には，形式的手法により，機械的に証明の各ステップの正しさの確認を行うことが望ましい．

従来より筆者の研究グループでは，順序機械型プログラムを対象とし，その正しさの形式的な証明を現実に可能とするための，代数的手法を用いた仕様記述法，設計法，検証法について研究してきた．我々の提案してきた方法では，要求仕様から実行可能プログラムまで段階的に詳細化し，各段階ごとに詳細化が正しいことを証明する[1]．各段階の仕様は，我々の代数的言語 ASL[2]のサブクラスである抽象的順序機械型 ASL/ASM[1]を用いて記述する．詳細化では，順序機械の抽象的な一遷移を，より具体的な動作（遷移）の系列で局所的に展開して下位レベルの仕様を得る．証明では，展開に用いた動作系列を下位レベルで実行した後に，展開した遷移の動作内容が満たされることを示す．

上述の設計法のもとでは，通常，展開する一遷移の動作はそれほど複雑ではなく，また展開に用いる動作系列もそれほど複雑ではないので，詳細化の正しさの証明は比較的容易である．これより，一般には困難なプログラムの正しさの証明が，現実に可能になることが期待される．

実用規模の設計例の検証が可能かどうか，検証にどの程度の手間がかかるのかを調べることが重要である．そこで，我々は従来，記述支援系・検証支援系などを試作し，クイックソートプログラム[3]や酒屋在庫管理プログラム[4]などの検証実験を行ってきた．

それらの検証例では，処理の要求性質を表す公理（すべての変数が式の頭で ∀ で束縛された等式）を，自由に変数を用いて記述していた．例えば，配列中のデータのソーティングを行う遷移 sort に対する要求の一つ「sort の実行後に，配列 ARY の $ArrayBottom$ と $ArrayTop$（これらは定数とする）の間のデータが昇順に並ぶこと」を，配列の任意の場所（インデックス）を表す変数 i を用いて

$$\forall i \, \forall s \, [\, ArrayBottom \leq i < ArrayTop \supset ARY(sort(s))[i] \leq ARY(sort(s))[i+1] == TRUE \,]$$

と書いていた．ここで，s は順序機械の抽象状態を表す特別な変数であり，この公理は，任意の抽象状態 s から遷移 sort を実行した後で，ARY が上述の関係を満たさなければならないということを表している．

証明では，sort の展開に用いた動作系列を下位レベルで実行した後で，上の等式で表される関係（以下，証明する式とも呼ぶ）が成り立つことを，動作系列の深さに関する構造的帰納法を適用したり，項書き換えや場合分けを証明する式に適用したりして示す[3,5]．しかし，実際の証明では，不変表明や下位レベルの公理の変数へどのような定数を代入するか，項書き換えでどの書き換え規則（公理）をどの順に用いるか，といった証明手順を検証者が考案するのがかなり難しく，検証支援系で対話的に項書き換えや場合分けなどを行う作業にも手間がかかった．

そこで，本論文では検証の自動化を進めるため，まず，証明する式が変数を持たない式となるように，仕様の記述スタイルを制限した．そのようにすることで，後述の一定の手順で証明が行え，証明手順の考案の必要がなくなる．

記述スタイルの制限は，「仕様中の各公理を，抽象状態の変数 s 以外の変数（先の例での i）を用いずに書く」というものである．証明時に，変数 s には帰納法の定数を代入するので，スタイル制限に従って仕様を記述していれば，証明する式は変数を持たない式になる．

この記述スタイル制限のもとでは，例えば先のソーティング sort に対する要求性質を，配列 a 中の p, q 間が昇順に並んでいることを表す述語 $Ordered(a,p,q)$ を基本述語として導入し，

$$\forall s \, [\, ArrayBottom \leq ArrayTop \supset Ordered(ARY(sort(s)), ArrayBottom, ArrayTop) == TRUE \,]$$

と書く．このように，集合（や配列）の全体を引数とし，ある性質が集合の各要素により満たされるとき真となる述語を導入することで，集合の任意の要素を表す変数を用いず，s だけを変数に用いて公理を記述できる．

本論文では，次に，上述の制限のもとで書かれた仕様に対する証明手順を考案した．それは以下のような方法である：まず (1) 検証者が不変表明と基本関数・述語に関する補題を考案した後，構造的帰納法の各段階ごとに，(2) その段階に対応する式を不変表明，各遷移の動作内容・実行順・実行条件の指定，基本関数・述語に関する補題より作り（上述のスタイル制限のもとでは，変数を持たない式となる），(3) 式中の "$+, -, >, =, \wedge, \vee, \neg, \supset$" はその解釈，他の関数・述語は自由解釈としたときに，(2) で作った式が成り立つことを，加減算・比較演算を持つ整数上の論理式（プレスブルガー文）の真偽判定手続き[6]を用いて示す．以上の証明手順は単純なため，検証をかなり自動化できる．

上述の記述スタイル制限のもとでは，上述の

Ordered のような抽象度の高い基本述語・関数を導入することになる．そうすると，証明では，それらの基本述語・関数に関する補題を一般に多く導入しなければならず，プレスブルガー文がかなり大きくなる場合がある．しかし，スタイル制限のもとでは，プレスブルガー文が，すべての変数が ∀ で束縛された冠頭標準形になるので，大きな式であっても実際上は十分高速に真偽判定を行える[7]（実例を4章で示す）．

このように，証明手順を検証者が考案する必要がないことと，証明の一回の試行を高速に行えることから，従来より短期間で証明を行えることが期待できる．しかし，実際にどの程度の手間で証明が行えるかを調べることが重要である．そこで，本論文では，先の証明法に基づく検証支援系を実際に作成し，例題の検証実験を行った．

本支援系を用いた検証作業では，検証者は不変表明と基本関数・述語の性質に関する補題（および補題中の変数への定数値の代入）をまとめて支援系に与え，上述の証明手順の実行を開始するコマンドを送るだけでよい．プレスブルガー文の合成や真偽判定などは本支援系が自動的に行う．

本論文では，例題として，マックスソートプログラムを用いた．この証明には，いわゆる「アルゴリズムの正しさ」の証明も含まれており，それ程簡単ではない（プログラムがマックスソートアルゴリズムを正しく実現しているという証明ではなく，そのプログラムがデータを並べ替えて停止するという証明を行っている）．検証では，まず，不変表明や基本関数・述語に関する補題を，数時間程度かけてある程度考案し，続いて，それらを実際に検証支援系に与えて証明を試みた．そのとき，事前に考案した不変表明中の条件の抜けなどにより十数回，証明失敗を繰り返した．しかし，それらの失敗を合わせても，検証作業をCPU時間十数分程度（SONY NEWS-5000使用，100MIPS）で高速に行えた．証明では，変数や演算子等の総出現回数が1300以上の，幾つかの大きなプレスブルガー文の真偽を判定しなければならなかったが，本支援系では，数秒程度で高速に判定できた．

この例題を含め，酒屋在庫管理プログラムの検証実験[8]や，ベンチマークCPUなど同期式順序回路の検証実験[9]を通して，本論文で提案する証明法により，従来より短期間で，かつ実用的と思われる時間で証明を行えるとの見通しを得た．在庫管理プログラムのように要求性質が比較的複雑な場合でも，導入する基本述語・関数をうまく決めれば，本論文で述べるスタイル制限のもとで仕様を自然に書ける場合があることも分かっている[8]．

以下，2章では従来より提案してきた代数的言語による順序機械型プログラムの仕様記述法と設計法，今回提案する仕様記述のスタイル制限，およびスタイル制限のもとでの仕様記述と設計例について，3章では今回考案した順序機械型プログラム向きの証明手順と作成した検証支援系の特徴的な機能について，4章ではその支援系を使って例題の検証実験を行った結果とその考察について述べる．

2. 順序機械型プログラムの記述法と設計法

2.1 仕様の記述法とスタイル制限

2.1.1 仕様の記述法

記述には，我々の研究グループが定義した代数的言語 ASL[2]のサブクラスである抽象的順序機械型 ASL/ASM[1]を用いる．

一般に代数的言語は，公理により表現式の等式集合を指定し，その等式集合から導かれる合同関係を，記述の意味定義としている．ASLでは，公理によって生成される等式集合から導かれる最小の合同関係を記述の意味定義とし，また表現式を定義するにあたって，関数の構文やソート間の包含関係を自然に定義できるように文脈自由文法を用いているのが，その特徴である[2]．

ASL/ASMによる記述（以下，ASM型記述と呼ぶ）では，抽象状態を表すデータ型を導入し，以下のような関数を用いる．(A) 状態成分関数は，抽象状態を引数とし，抽象状態におけるプログラム変数の値を表す．(B) 状態遷移関数は，抽象状態を引数とし，遷移後の抽象状態を表す．(C) ループ関数は，抽象状態を引数とし，複数の状態遷移関数および他のループ関数間の部分的な実行順序を指定する．

ASM型記述は，一般に，有限個のレジスタ（データ型は任意）と（有限状態）制御部を持つレジスタ付き状態機械を代数的に記述しているものと思ってよい．具体的には，各遷移の動作内容と，各遷移の実行順序・実行条件（以下，実行指定とも呼ぶ）を公理を用いて指定する．公理は，変数を用いて $p(v_1, v_2, \cdots) == q(v_1, v_2, \cdots)$ の形で書かれた等式のことで，述語論理での $\forall v_1 \forall v_2 \cdots [p(v_1, v_2, \cdots) = q(v_1, v_2, \cdots)]$ という式に相当する．

一つの遷移 T の動作内容の公理では，抽象状態の特別な変数 s を用いて，任意の状態 s に対して，T の実行前の状態 s での各状態成分関数の値と，実行後の状態 $T(s)$ での各状態成分関数の値が，どのような関係を満たすべきかを記述する（一般に，複数個の公

理を用いる).いま,あるレベルの記述において p 個の状態成分関数が使われているとする.状態成分関数 F_i $(1 \leq i \leq p)$ の,遷移 T の実行前後での値の関係を

$$F_i(T(s)) == C_i(F_1(s), \cdots, F_p(s))$$

のように,T の実行後の F_i の値が実行前の各 F_j $(1 \leq j \leq p)$ で定まるように記述すれば,その公理を代入定義と呼ぶ.また,各状態成分の値の関係を

$$P_k(F_1(s), \cdots, F_p(s), F_1(T(s)), \cdots, F_p(T(s))) == TRUE$$

のように,T の実行前後の状態成分値を引数とするような述語 P_k で指定するときは,その公理を性質記述と呼ぶ.

実行指定は,ループ関数 (L) を使用し,以下のいずれかの形の公理で,末端再帰だけを用いて書く☆.

(C-1) $L(s) ==$ if *bool* then LT else LT'

(C-2) $L(s) == LT$

ただし L' を他のループ関数として,

LT, LT' は $L'(T(s)), T(s), s$ のいずれか.

2.1.2 仕様記述スタイルの制限

今回,全レベルの仕様で,遷移の動作内容の公理と実行指定の公理を,以下のスタイル制限のもとで書くことにした.

遷移の動作内容の記述スタイル制限:抽象状態の変数 s 以外の変数を用いずに記述する.

実行指定の記述スタイル制限:(C-1) での if 文の条件式(遷移の実行条件)*bool* を,s 以外の変数を用いずに記述する.

これらの制限は,1章でも述べたように,証明する式を変数を持たない式にすることを,その目的としている.

2.2 詳細化による設計法

我々の提案する階層的設計法では,抽象度の高いレベルから実行可能なプログラムのレベルまで,逐次詳細化を行う[1].詳細化では,第 k レベルに性質記述を用いて動作内容が指定された遷移 T があるとき,それを第 $k+1$ レベルで,より具体的な遷移 $t_1, \cdots, t_n$ を用いて実現する.このために,遷移 $t_1, \cdots, t_n$ の動作内容を性質記述や代入定義を用いて指定し,T をループ関数として $t_1, \cdots, t_n$ の実行順と実行条件を指定する.必要であれば,第 $k+1$ レベルで新しい状態成分関数を幾つか導入する.第 k レベルの複数の遷移 $T_1, \cdots, T_m$ を,第 $k+1$ レベルで同時に詳細化してもよい.第 $k+1$ レベルの記述は,第 k レベルの記述中の公理のうち,遷移 T の動作内容の公理以外のものをすべて含む.

2.3 スタイル制限のもとでの仕様記述と詳細化による設計の例

ここでは,整数の配列 *ARY* のうち,ポインタ(整数型の状態成分関数)*LowerBound*, *UpperBound* で指される間の領域のソーティングを行うプログラムの仕様を 2.1.2 項の制限のもとで記述し,詳細化により設計した例を示す.

ArrayTop を配列の上限,*ArrayBottom* を下限を表すフォーマルパラメータ☆☆とし,プログラムの実行前に,$ArrayBottom \leq LowerBound(s) \leq UpperBound(s) \leq ArrayTop$ が成り立っているとする.導入するすべての基本関数について,その引数となる配列は,その上限が *ArrayTop*,下限が *ArrayBottom* であるとする.

処理の要求記述(レベル1.**図1**参照)では,一回の実行でソーティングを行う遷移 sort の動作内容を,配列全体を引数とする基本述語 *Ordered*, *SameSet* を導入して記述した.このような配列全体を引数とする基本述語を用いることで,配列の任意の場所を表す変数を使用せず,2.1.2 項での記述スタイル制限に従って仕様を書くことができる.なお,図1の記述例では,「ソートする範囲の外側は不変」という要求は課していないが,後述の述語 *SameArray* を用いて書き,要求に加えることもできる.

レベル2ではレベル1の遷移 sort を,マックスソートアルゴリズムで,すなわち,「配列中の未ソートの領域中の最大値を線形探索により発見して,ソート済みの領域の隣のデータと交換する」動作の繰り返しで実現した.このために,配列のうち未ソートの領域とソート済みの領域の境界を指すポインタ *bound* (*bound*+1 から *UpperBound* までがソート済み),最大値の検索のとき最大値と比較する場所を指すポインタ *arypnt*,最大値の存在箇所を指すポインタ *maxpnt* を新しい状態成分として導入した.次に,各ポインタの設定や配列中のデータの交換等を行う遷移を幾つか導入し,各遷移の動作内容を基本述語 *SameArray*, 基本関数 *iget*(これらも新たに導入した)を用いて記

☆ 本論文では,これらの公理だけを用いて実行制御を書ける ASM 型記述だけを対象とする.そのような記述より得られるプログラムでは,実行時にシステムスタックが不要なため,証明を,状態遷移関数の深さに関する構造的帰納法(3.3.1 項で説明する)で行える.

☆☆ C 言語等へ変換する際などに,整数定数を代入する.証明の際には,*ArrayTop*, *ArrayBottom* は任意の値をとるものとして扱う(配列 *ARY* が任意の大きさをとる場合について,ソーティングが正しく行われることを証明する).

導入した基本述語：

● Ordered(ary,p1,p2)

配列 ary の p1～p2（p1 ≦ p2）のデータが昇順に並んでいるとき真となる述語．

● SameSet(ary1,p1,p2,ary2,p3,p4)

配列 ary1 の p1～p2（p1 ≦ p2）のデータと，配列 ary2 の p3～p4（p3 ≦ p4）のデータが多重集合として等しいときに真となる述語．

```
define 'sort 前提条件'
     := 'ArrayBottom ≦ LowerBound(s)
                   ≦ UpperBound(s) ≦ ArrayTop';

性質 SORT1:
(* LowerBound ~ UpperBound が整列される *)
 sort 前提条件
 imply Ordered(ARY(sort(s)),LowerBound(s),
                      UpperBound(s)) == TRUE;

性質 SORT2:
(* LowerBound ~ UpperBound は多重集合として不変 *)
 sort 前提条件
 imply | [ LowerBound(s) = LowerBound(sort(s))
          and UpperBound(s) = UpperBound(sort(s)) ]
     imply SameSet(ARY(s), LowerBound(s),
                         UpperBound(s),
                ARY(sort(s)), LowerBound(s),
                         UpperBound(s)) |
 == TRUE;
```

図 1　レベル 1 の遷移 sort に対する要求性質
Fig. 1　Requirement description for the transition 'sort' in level 1.

導入した基本述語・関数

● SameArray(ary1,p1,p2,ary2,p3,p4)

配列 ary1 の p1～p2（p1 ≦ p2）と，配列 ary2 の p3～p4（p3 ≦ p4）で，データの値と並び方が全く同じであるとき，真となる述語．
p1～p2 の幅と，p3～p4 の幅は異なっていてもよい．

● iget(ary,p)

配列 ary の p で示される箇所の値を返す関数．
この関数は ASL コンパイラにより用意されている．

```
(*** 各遷移の動作内容 ***)
(*** ここでは，状態成分の値が変わらないことの記述は省略 ***)

bound(initAll(s))     == UpperBound(s);
maxpnt(initMax(s))    == LowerBound(s);
arypnt(initMax(s))    == LowerBound(s) + 1;
maxpnt(setMax(s))     == arypnt(s);
arypnt(setMax(s))     == arypnt(s) + 1;
arypnt(unsetMax(s))   == arypnt(s) + 1;

(** 遷移 nop の実行では，すべての状態成分が不変 **)

(** 遷移 exchange の動作内容：
   実行後，maxpnt(s) の指す箇所のデータと bound(s) の指す
   箇所のデータが交換される **)

define 'exchange 前提条件' :=
     'ArrayBottom ≦ maxpnt(s) ≦ ArrayTop
      and ArrayBottom ≦ bound(s) ≦ ArrayTop';

 bound(exchange(s)) == bound(s) - 1;

性質 exg1:
(* bound(s) の指す箇所に
  maxpnt(s) の指す箇所のデータが入る *)
 exchange 前提条件
 imply iget(ARY(exchange(s)),bound(s))
        = iget(ARY(s),maxpnt(s)) == TRUE;

性質 exg2:
(* maxpnt(s) の指す箇所に
  bound(s) の指す箇所のデータが入る *)
 exchange 前提条件
 imply iget(ARY(exchange(s)),maxpnt(s))
        = iget(ARY(s),bound(s)) == TRUE;

性質 exg3:
(* bound(s),maxpnt(s) の指す箇所以外のデータは不変 *)
 exchange 前提条件
 imply |
    [ maxpnt(s) ≦ bound(s)
      imply (   ( ArrayTop ≦ maxpnt(s)-1
              imply SameArray(ARY(s),ArrayBottom,
                               maxpnt(s)-1,
                     ARY(exchange(s)),ArrayBottom,
                               maxpnt(s)-1) )
          and ( maxpnt(s)+1 ≦ bound(s)-1
              imply SameArray(ARY(s),maxpnt(s)+1,
                               bound(s)-1,
                     ARY(exchange(s)),maxpnt(s)+1,
                               bound(s)-1) )
          and ( bound(s)+1 ≦ ArrayTop
              imply SameArray(ARY(s),bound(s)+1,
                               ArrayTop,
                       ARY(exchange(s)),bound(s)+1,
                               ArrayTop) )
    ]
 and [ bound(s) < maxpnt(s)
      imply （上の maxpnt と bound が逆になった式）
    ]
| == TRUE;
```

図 2　レベル 2 の遷移の動作内容
Fig. 2　Contents of transitions in level 2.

述した（**図 2** 参照）．次に，それらの遷移の実行順・実行条件を指定した（**図 3** 参照）．

本来，レベル 2 では「配列中の未ソートの領域中の最大値を見つけ，その最大値とソート済みの領域の隣のデータを交換する」動作を一つの遷移で行うことにしておき，それをレベル 3 以降で複数の遷移によって実現する方が自然である．この例では，詳細化で複雑な系列を用いて展開しても，比較的容易に検証作業が行えることを示す（4 章で述べる）ために，一回の詳細化でほとんどの処理を実現した．

レベル 3 では，遷移 exchange を，基本関数 *iput* を用いて実現した（**図 4** 参照）．レベル 3 までで，すべての遷移の動作内容が代入定義で記述され，かつ代入値の計算に用いる基本関数（*iget*, *iput*）がすべて ASL コンパイラによって用意されたものになったため，C 言語等に変換可能となる[1]．

3. 順序機械型プログラム向きの証明手順と検証支援系

ここでは，検証の自動化を進めるために今回考案した証明手順と，その手順に基づいて作成した検証支援系の機能について述べる．

3.1 詳細化の正しさの定義

項の組 $\langle \xi, \eta \rangle$ がテキスト t のもとで定理として成り立つのは，ξ, η の変数への許される任意の（変数を含まない）代入 ρ に対して $\rho\xi \equiv_t \rho\eta$ が成り立つこと，すなわちテキスト t 上の帰納的定理であることとし，これを $\xi \approx_t \eta$ と表すことにする[2]．

上位のテキストで公理として表された各等式 $\xi == \eta$

```
sort(s) == S1(initAll(s));

S1(s) ==  if LowerBound(s) < bound(s)
          then S2(initMax(s))
          else nop(s);

S2(s) ==  if arypnt(s) ≦ bound(s)
          then if  iget(ARY(s),maxpnt(s))
                       < iget(ARY(s),arypnt(s))
              then S2(setMax(s))
              else  S2(unsetMax(s));
          else S1(exchange(s));
```

図3 レベル2の遷移の実行指定
Fig. 3 Execution order of transitions in level 2.

```
導入した基本関数

● iput(ary,p,d)

  配列 ary の p で示される箇所に d の値を代入した配列を返す関数.
  この関数は ASL コンパイラにより用意されている.

(** 状態成分の値が変わらないことの記述は省略 **)
(* 遷移 exchange の動作内容 *)

bound(exchange(s)) == bound(s)-1;

ARY(exchange(s)) ==
  iput( iput( ARY(s),maxpnt(s),iget(ARY(s),bound(s)) ),
        bound(s),iget(ARY(s),maxpnt(s)) );
```

図4 レベル3の遷移の動作内容
Fig. 4 Contents of transitions in level 3.

に対して，下位のテキスト t の公理系のもとで $\xi \approx_t \eta$ が成り立つとき，詳細化は正しいとする[3]．厳密には下位の公理が無矛盾であることも必要であるが，今回は議論しない☆.

$\xi \approx_t \eta$ を示すには，項書き換え，場合分け，整数上の決定問題への帰着，補題の利用，帰納法などを用いる[3]．これらの証明法自体は健全である．

3.2 不変表明などの記述スタイルの制限

今回，2.1.2 項で述べた仕様の記述スタイル制限に加え，構造的帰納法で用いる状態に対する不変表明 Inv，および停止性の証明で用いる関数 $f_{terminate}$（これらについては 3.3.1 項で説明する）の記述スタイルと，基本関数・述語に関する補題（これについては本節の最後で説明する）を証明で使用する方法も，それぞれ以下のように制限した．制限の目的は，仕様の記述スタイル制限と同様，証明する式を変数を持たない式にすることである．

状態に対する不変表明と停止性の証明で用いる関数の記述スタイル制限：いずれも抽象状態の変数 s 以外の変数を使用せずに記述する．

基本関数・述語に関する補題を証明で使用する方法についての制限：基本関数・述語に関する補題中のすべての変数へ値を代入して得られるインスタンスだけを，証明で使う．代入する値は，定数，定数を引数とする関数・述語，および，それらのみを引数に持つ関数・述語のいずれかとし，インスタンスが変数を持たない式となるようにする．

ここで，基本関数・述語に関する補題とは，基本関数・述語に関して常に成り立つ（と思われる）性質のことで，検証者が無条件に正しいものとして導入する．補題は，公理（等式）の形で書くが，その記述スタイルは特に制限しない．

3.3 順序機械型プログラム向きの証明手順

以下，上位レベルの遷移 T を，下位レベルの遷移系列 L で展開したとする．ここでは，遷移 T の動作内容として書かれた一つの公理（等式）

$$Pre(s)\ imply\ Post(s,T(s)) == TRUE$$

（代入定義，性質記述のいずれで書かれていても，上のような一般形で表せる）が，下位レベルのテキスト上で定理となることを証明するための，今回考案した証明手順を説明する．

3.3.1 順序機械型プログラムの証明で示す事項と証明に用いる手法

証明で示すことは，(a) T の展開に用いた下位レベルの遷移系列 L の実行前での状態成分値（それらは述語 Pre を満たすと仮定する）と，実行後での状態成分値（もし定まるなら）との間に，述語 $Post$ で表された関係が満たされること（部分正当性）と，(b) L の実行後の状態成分の値が定まること（停止性）である[5]．うち (b) については，以下の二つを示せばよい：(b-1) L に遷移の繰り返し実行があれば，その実行が必ず終了すること．(b-2) L の各遷移の動作内容や実行条件の記述で用いられている関数・述語が，その引数が L の実行中にとり得る任意の値に対して，値を持つ（その関数の各引数の値が，定義域内にある）こと．

以下，これらを証明するために用いる手法について述べる．

(a) の証明は，L が一般に閉路（遷移の繰り返し実行）を持つので，L 中の遷移 $t_1,\cdots,t_n$ の実行回数（状態遷移関数のネスティングの深さ）に関する構造的帰納法[5]を用いて行う．

構造的帰納法による証明の方法について，簡単に述べる．まず，L 上の適当な（幾つかの）状態 s に対して不変表明 $Inv(S0,s)$ を与える．不変表明を与える状態は，不変表明を持たない状態だけを通る閉路が存在しなくなるように選ぶ．また，L の実行が終了した時

☆ プログラムレベルでは，性質記述ではなく代入定義の形の公理で書かれるので，その形より無矛盾は保証される．したがって，各レベルの詳細化が正しければ，各レベルの記述も無矛盾となる．

点（状態）にも不変表明を与え，それは $Post(s,T(s))$ に対する式 $Post(S0,s)$ とする．

ここで，$S0$ は L による遷移を始める直前の抽象状態を表す特別な定数であり（その時点での状態成分 F_i の値を $F_i(S0)$ で参照する），不変表明には，その抽象状態 s での状態成分値（$F_j(s)$ で参照する）と $F_i(S0)$ 等との関係を書く．

続いて，(i) $S0$ から，任意の L 上の遷移系列（以下パスと呼ぶ）を実行して，不変表明を持つ状態に到達した場合，(ii) 不変表明を持つ状態から任意のパスを実行して，不変表明を持つ状態に到達した場合，および (iii) $S0$ あるいは不変表明を与えた抽象状態から任意のパスを実行して，L の実行が終了した場合のそれぞれについて，「そのパスの終点での不変表明が，$Pre(S0)$，パスの始点での不変表明（あれば），L の各遷移の動作内容・実行制御の指定，および基本関数・述語に関する補題のもとで成り立つ」ことを証明する．

(b-1) については，上の (ii) のパスの実行の度に減少し，かつ実行中は非負の値をとるような（整数型の）関数 $f_{terminate}(S0,s)$ を見つけ，そのことを証明する[5]．

(b-2) については，例えば以下のような方法で保証できるが，本論文では陽に議論しない：「各基本関数・述語の引数（状態成分など）の値が，その関数の定義域内にある」ことを表す条件式を，(a) で用いる各状態に対する不変表明に入れ，かつ，基本関数・述語の各補題の前提条件にも入れる．このようにすれば，L の実行中に引数の値が関数の定義域外の値を取る場合，(a) の証明ができなくなる．

(a), (b) の証明がすべて成功すれば，検証者が導入した基本関数・述語に関する補題が正しいという前提のもとで，$Pre(s)\ imply\ Post(s,T(s)) == TRUE$ が定理として成り立つと結論する．

3.3.2 今回考案した証明手順

3.3.1 項における，(a) での構造的帰納法の各パスの証明，および (b-1) での，1 回のパスの実行で関数値が減少し，かつ実行中は関数の値が 0 以上であることの証明は，いずれも同一の手順で証明できる．

今回，それらの証明を行う手順を考案した．以下，(a) におけるパス $t_a \cdots t_b$ の証明を例に，考案した証明手順を説明する．ここで，各遷移は実行条件 $cond_{t_a}(s)$，$\cdots$，$cond_{t_b}(s)$ を持ち，パスの始点での不変表明が $Inv_1(S0,s)$，パスの終点での不変表明が $Inv_2(S0,s)$ であるとする．

Step (a)： 式 $P_1 \triangleq$

$$[\ Pre(S0)\ \wedge Inv_1(S0,S) \wedge cond_{t_a}(S) \wedge \ldots \wedge cond_{t_b}(\cdots(t_a(S))\cdots)\ ] \supset Inv_2(S0, t_b(\cdots(t_a(S))\cdots))$$

を作る．

これは，「パスの実行前に，$Pre(S0)$，$Inv_1(S0,S)$，およびパスの実行条件が成り立つならば，パスの実行後に，$Inv_2(S0,t_b(\cdots(t_a(S))\cdots))$ が成り立つ」ことを表す式である．ここで，S は構造的帰納法で用いる抽象状態の定数である．

なお，(b-1) の証明で，例えばパスの実行により $f_{terminate}$ の値が減少することを証明する場合は，式 $P_1 \triangleq$

$$[\ Pre(S0)\ \wedge f_{terminate}(S0,S) > 0 \wedge cond_{t_a}(S) \wedge \ldots \wedge cond_{t_b}(\cdots(t_a(S))\cdots)\ ] \supset f_{terminate}(S0,S) > f_{terminate}(S0,t_b(\cdots(t_a(S))\cdots))$$

とすればよい．

いずれの場合も，2.1.2 項と 3.2 節の制限のもとでは，P_1 は変数のない式となる．その理由は，不変表明 $Inv(S0,s)$，$Pre(S0)$，$Post(S0,s)$，$f_{terminate}(S0,s)$，遷移の実行条件 $cond(s)$ のいずれも s 以外の変数を持たず，かつ，その変数 s には定数 S または $S0$ を代入するからである．

Step (b)： パス上の各遷移の動作内容の公理のうち，代入定義の形をした各公理について，その公理のインスタンス（パス $t_a \cdots t_{k-1} t_k \cdots t_b$ 上の遷移 t_k の公理については，その $==$ を $=$ に置き換え，変数 s に定数 $t_{k-1}(\cdots(t_a(S))\cdots)$ を代入して得られる式）を P_1 の前提とするか，または，その公理を左辺から右辺への書き換え規則とみなして P_1 を書き換える（項書き換えの適用）．続いて，性質記述の形をした各公理のインスタンス（代入定義の場合と同様に，$==$ を $=$ に置き換え，s に定数 $S, t_a(S), \cdots$ を代入して得られる式）を書き換え後の式の前提にする．以上により得られた式を P_2 とする．

ここで，2.1.2 項の制限のもとでは，得られる式 P_2 は変数を持たない式となる．その理由は，以下の二つである：(i) 項書き換えを行う際，書き換える式 P_1 は変数を持たない式であり，かつ書き換え規則の形（変数が s だけである）より，書き換え後の式も変数を持たない式になること．(ii) 式の前提にする公理のインスタンスは，もとの公理が s 以外の変数を持たないので，変数を持たない式であること．

代入定義の形の公理を，書き換え規則として用いるか前提とするかは，検証者が自由に決定して良い．どちらでも証明は行える．項書き換えを行う場合は，式 P_2 が簡単になり，必要な基本関数・述語の補題は少

なくてすむが，反面，項書き換えがどのように行われるかを予想したうえで，補題中の変数への代入値を決定しなければならないので，代入値の考案がやや難しくなる．前提として用いる場合は，逆に，基本関数・述語の補題は多く必要となるが☆，代入値の考案は比較的容易である．

Step (c)： 基本関数・述語の補題のインスタンスを P_2 の前提条件とした式 P_3 を作る．

3.2 節の制限のもとでは，P_3 は変数のない式となる．その理由は，P_2 の前提にする式（補題のインスタンス）が，補題中のすべての変数へ定数を代入して得られた，変数を持たない式だからである．

Step (d)： P_3 中の記号 "$+, -, >, =, \wedge, \vee, \neg, \supset$"（整数の加減算・比較演算，および論理演算）にはさまれているすべての部分項を，それぞれ整数型・論理型の変数に置き換える．ただし，文字列として同じ部分項は同じ変数に，異なる部分項は異なる変数に置き換える．得られた式を P_4 とする．

例えば，P_3 を

$$(r1(S)+r2(S)=r1(S)\times r2(S)\vee pred(r1(S))\wedge\cdots)$$
$$\supset\ (r1(S)>0\vee\cdots)$$

とすれば，$r1(S), r2(S), r1(S)\times r2(S), pred(r1(S))$ をそれぞれ v_1, v_2, v_3, v_4 （v_1, v_2, v_3 は整数型の変数，v_4 は論理型の変数）に置き換えた

$$(v_1+v_2=v_3\vee v_4\wedge\cdots)\ \supset\ (v_1>0\vee\cdots)$$

が P_4 である．

2.1.2 項と 3.2 節の制限のもとでは，（P_3 が変数を持たない式となるので）P_4 は整数型または論理型の自由変数だけを持つ式となる．

Step (e)： P_4 中の変数をすべて式の頭で ∀ で束縛した式 P_5 を作る．

P_4 は自由変数だけを持つ式であったので，P_5 は，その作り方から，すべての変数が ∀ で束縛された冠頭標準形の式となる．

かつ，P_5 は，変数が整数型または論理型で，演算が "$+,-,>,=,\cdots$" だけの閉論理式，すなわちプレスブルガー文となり☆☆，真偽が決定可能である[6]．

☆ 例えば代入定義 $F(t(s)) == F(s)$ があり P_1 が $[\cdots]\supset func(F(t(S)))=func(F(S))$ であった場合など，項書き換えをすれば P_2 は $[\cdots]\supset func(F(S))=func(F(S))$ となり，この式の成立は関数 $func$ に関する補題を導入しなくても示せる．一般に項書き換えをしないと，証明で，各基本関数・述語について，「引数が同じ値なら，その関数・述語の値も同じである」ことを表す補題などが必要となってくる．

☆☆ 文献 6) 中での定義とは異なり，本論文では，整数型の変数だけでなく論理型の変数を持つ式も，プレスブルガー文と呼ぶ．論理型の変数があっても，その取り得る値が true または false と有限個なので，真偽は決定可能である．

Step (f)： P_5 の真偽を，プレスブルガー文真偽判定手続きで判定し，結果が真であれば，そのパスの証明は成功（そのパスの実行後に Inv_2 が成り立つ）と結論する．

以上が一つのパスの証明の手順である．これを各パスごと，すべてのパスの証明が成功するまで繰り返す．あるパス P の証明が失敗した（Step (f) で，判定結果が偽となった）場合は，不変表明の修正，証明で用いる基本関数・述語に関するインスタンスの追加，設計の修正を適宜行った後，そのパス P の再証明を行う．不変表明や設計を修正した場合，P 以外のパスについても再証明が必要になる．

以下，2.1.2 項や 3.2 節での制限が満たされなかった場合の問題点について述べる．そのような場合，例えば仕様や不変表明が s 以外の変数を用いて書かれていた場合には，Step (c) で P_3 が

$$\forall v_{11}\forall v_{12}\cdots\{\ [Pre(S0,v_{11},v_{12},\cdots)$$
$$\wedge(\forall v_{21}\forall v_{22}\cdots Inv_1(S0,S,v_{21},v_{22},\cdots))$$
$$\wedge\cdots]$$
$$\supset\ Post(S0,t_b(\cdots t_a(S)\cdots),v_{11},v_{12},\cdots)\ \}$$

というような，束縛変数を持つ式になってしまう．そのような式が成り立つことは，Step (d)～(f) の方法では通常は示せないので，上述の証明手順では証明を行えなくなってしまう．

3.4 本検証支援系の特徴的な機能

今回，3.3 節で述べた証明法に基づく新しい検証支援系を作成した．本支援系の特徴としては，検証者が証明の際に，上位・下位レベルの仕様中の各公理，不変表明，基本関数・述語に関する補題，および補題の各変数への代入値を支援系に与えるだけでよく，補題のインスタンスの合成や，3.3.2 項の Step (a)～(f) の実行を支援系が自動で行うことが挙げられる．

以下，この他に本支援系が持つ幾つかの特徴的な機能と，それらの目的などについて述べる．

機能 1： 下位レベルの遷移の実行順を状態図で表示する機能（**図 5** 参照）．

この状態図は本支援系のユーザインタフェースとなっており，検証者は不変表明を与える状態の指定を容易に行える．不変表明の設定や修正を直観的に分かりやすい方法で行えるようにするため，このような機能を設けた．

また，本支援系は，各パスの証明が終了しているかどうかの自動管理を行い（不変表明の修正時に，適当なパスを未証明に戻すなどの操作も行う），未証明のパスがどれか等の情報を状態図中に表示できる．

機能 2： 基本関数・述語の補題の変数に対する代入値

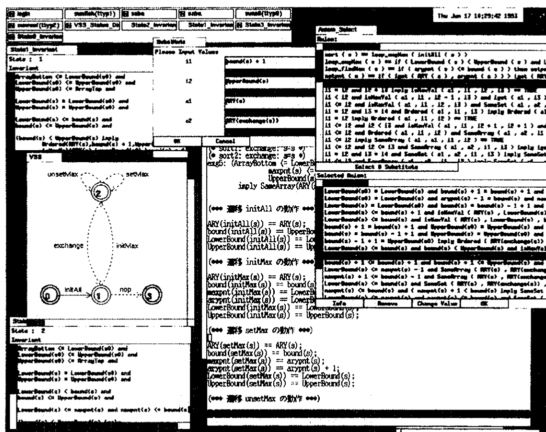


図5 検証支援系
Fig. 5 Display of our verification support system.

の入力支援機能.

本支援系は，基本関数・述語の補題の各変数への代入値の候補を幾つか求めて，それらの一覧表（メニュー）を検証者に提示する機能を持つ．各パスの証明における式 P_3 と基本関数・述語の補題との最汎単一化子(mgu)が，代入値の候補となる．検証者が，その一覧表から代入値を選択すると，本支援系は，選択された代入値と補題の式から，自動でインスタンスの式を合成する．多くの場合，この機能は，検証者が考案した代入値を，候補の一つとして提示できる．

また，本支援系では，得られた基本関数・述語に関する補題のインスタンスを，すべてのパスの証明に共通に用いる（3.3.2 項の Step (c) で P_2 の前提にする）ことも，どのパスの証明で使用するかを検証者がインスタンスごとに指定することもできる．いずれの方法をとるかは，検証者が指定する．前者の方が，支援系の操作が少なくてすむ．その場合，個々のパスの証明では余分なインスタンスが用いられることになり，真偽を判定するプレスブルガー文 P_5 が大きくなるが，それでも通常は十分な速度で証明できる（実例を 4 章で示す）.

機能3： 高速なプレスブルガー文の真偽判定機能.

3.3.2 項の Step (f) でプレスブルガー文の真偽を判定するが，本支援系では，判定する文が $\forall x_1 \forall x_2 \cdots P(x_1, x_2, \cdots)$ の形（すべての変数が $\forall$ で束縛された冠頭標準形）をしていることを利用し，真偽判定ルーチンに高速化の工夫をしている.

本支援系の判定ルーチンでは，式 $\exists x_1 \exists x_2 \cdots P'(x_1, x_2, \cdots)$ 中の変数を一つずつ消去（消去する変数を x として，x を含まない等価な式に変形）していくという判定アルゴリズム[6]を用いているが，高速化のために，変数の消去順をうまく決定する（式の構文木の根に近い変数から消去するなど）処理を追加している[7].

4. 検証支援系の評価実験と考察

4.1 マックスソートプログラムの検証実験

3.3.2 項で述べた証明手順により実際の例題の証明が行えるかどうか，証明にどの程度の手間がかかるか等を調べることが重要である．そこで，2.3 節で述べたマックスソートプログラムの，レベル 1 から 2 への詳細化の正しさと，レベル 2 から 3 への詳細化の正しさを，それぞれ 3.4 節で述べた検証支援系を用いて証明した.

まず，レベル 1–2 間の部分正当性（性質 SORT1, SORT2 で表された関係が，それぞれ下位レベルで成り立つこと）および停止性の証明において，用いた不変表明や基本関数の補題，および，証明にかかった手間について述べる.

部分正当性の証明に用いた不変表明や基本関数・述語に関する補題を，**図 6**，**図 7** にそれぞれ示す.

不変表明は，展開に用いた遷移系列上の閉路（図 3 でのループ関数 S1，S2 に相当する）の始点二つに与えた．性質 SORT1 の証明と SORT2 の証明では，それぞれ異なる不変表明を用いた．これらの不変表明には，各 S1，S2 において配列 ARY のどの部分がソート済みになっているか，また，各ポインタの位置関係がどのようになっているかなどが書かれている（不変表明の記述のために基本述語 $isMaxPos$ を導入している．図 6 参照).

導入した基本関数・述語に関する補題は，基本述語 $Ordered$ や $isMaxPos$ の再帰的定義の一部や，それらの述語（関数）間で成り立つ関係などである．SORT1 の証明では 40 個，SORT2 では 10 個程度と，かなり多くの基本関数の補題のインスタンスを使用した.

補題の変数への代入値は，例えば，図 7 の prim 2 の変数 a1 に対しては $ARY(exchange(S))$，i1 に対しては $bound(S)+1$，i2 に対しては $UpperBound(S)$ とした（これらの代入値を支援系に与えるとき，3.4 節で述べた代入値入力支援機能を用いた）．この代入により得られたインスタンスは，SORT1 の証明で，S2 から S1 に至るパスの実行後に，配列中のソート済みの範囲が一つ広がることを示すために用いた.

以上の不変表明，基本関数・述語に関する補題，および補題の変数への代入値を本検証支援系に与えることで，部分正当性の証明を成功させることができた.

導入した基本述語

● isMaxPos(array,pointer1,pointer2,pointer3)

pointer1 ≦ pointer2 で，かつ，配列 array の pointer3 で指される箇所の値が，pointer1～pointer2 中のデータの最大値以上のとき，真となる述語．

<< SORT1 の証明で用いた不変表明>>

```
 状態 S1 に対する表明：
   ArrayBottom ≦ LowerBound(S0) = LowerBound(s)
       ≦ bound(s) ≦ UpperBound(S0) = UpperBound(s)
       ≦ ArrayTop
 and ( bound(s) < UpperBound(s)
  imply ( Ordered(ARY(s),bound(s)+1,UpperBound(S0))
          and isMaxPos(ARY(s),
                 LowerBound(S0),
                 bound(s)+1,bound(s)+1) ) )


 状態 S2 に対する表明：
   ArrayBottom ≦ LowerBound(S0) = LowerBound(s)
       < bound(s) ≦ UpperBound(S0) = UpperBound(s)
       ≦ ArrayTop
 and LowerBound(s) ≦ maxpnt(s) ≦ bound(s)
 and LowerBound(s) < arypnt(s) ≦ bound(s)+1
 and isMaxPos(ARY(s),LowerBound(S0),
            arypnt(s)-1,maxpnt(s))
 and ( bound(s) < UpperBound(s)
  imply ( Ordered(ARY(s),bound(s)+1,UpperBound(S0))
          and isMaxPos(ARY(s),LowerBound(S0),
                 bound(s)+1,bound(s)+1) ) )
```

<< SORT2 の証明で用いた不変表明>>

```
 状態 S1 に対する不変表明
   ArrayBottom ≦ LowerBound(S0) = LowerBound(s)
       ≦ bound(s) ≦ UpperBound(S0) = UpperBound(s)
       ≦ ArrayTop
 and SameSet(ARY(S0),LowerBound(S0),
                  UpperBound(S0),
          ARY(s), LowerBound(S0),
                  UpperBound(S0))

 状態 S2 に対する不変表明
   ArrayBottom ≦ LowerBound(S0) = LowerBound(s)
       ≦ maxpnt(s) ≦ bound(s) ≦ UpperBound(S0)
       = UpperBound(s) ≦ ArrayTop
 and LowerBound(S0) < arypnt(s) ≦ bound(s)+1
 and LowerBound(S0) ≦ bound(s)
 and SameSet(ARY(S0),LowerBound(S0),
                  UpperBound(S0),
          ARY(s), LowerBound(S0),
                  UpperBound(S0))
```

図6 レベル 1−2 間の証明で使用した不変表明

Fig. 6 Invariant assertions for the proof between level 1 and level 2.

```
prim1:
(* 要素が 1 個の領域は Ordered *)
  ArrayBottom ≦ i1 = i2 ≦ ArrayTop
  imply Ordered(a1,i1,i2) == TRUE;

prim2:
(* Ordered な領域を左へ 1 つ拡大する *)
  ( ArrayBottom ≦ i1 < i2 ≦ ArrayTop
     and Ordered(a1,i1+1,i2)
     and iget(a1,i1) ≦ iget(a1,i1+1) )
  imply Ordered(a1,i1,i2) == TRUE;

prim3:
(* 要素が 1 個なら，それは最大値 *)
  ArrayBottom ≦ i1 = i2 = i3 ≦ ArrayTop
  imply isMaxPos(a1,i1,i2,i3) == TRUE;

prim4:
(* isMaxPos の領域を右へ 1 つ拡大する *)
  ( ArrayBottom ≦ i1 < i2 ≦ ArrayTop
     and ArrayBottom ≦ i3 ≦ ArrayTop
     and isMaxPos(a1,i1,i2-1,i3)
     and iget(a1,i2) ≦ iget(a1,i3) )
  imply isMaxPos(a1,i1,i2,i3) == TRUE;

prim5:
(* 要素数 1 の領域どうしはデータが同じなら SameSet *)
  ( ArrayBottom ≦ i1 ≦ ArrayTop
     and ArrayBottom ≦ i2 ≦ ArrayTop
     and iget(a1,i1) = iget(a2,i2) )
  imply SameSet(a1,i1,i1,a2,i2,i2) == TRUE;

prim6a:
(* 二つの領域が SameSet なら，各領域の端に
   それぞれ同じデータを付け足しても SameSet *)
  ( ArrayBottom < i1 ≦ i2 ≦ ArrayTop
     and ArrayBottom ≦ i3 ≦ i4 < ArrayTop
     and SameSet(a1,i1,i2,a2,i3,i4)
     and iget(a1,i1-1) = iget(a2,i4+1) )
  imply SameSet(a1,i1-1,i2,a2,i3,i4+1) == TRUE;
  など


prim7a: (* 領域 a と b, c と d が，それぞれ SameSet なら，
 a と c をマージした領域と，b と d をマージした領域も
 SameSet *)
  ( ArrayBottom ≦ i1 < i2 ≦ i3 ≦ ArrayTop
     and ArrayBottom ≦ i4 < i5 ≦ i6 ≦ ArrayTop
     and SameSet(a1,i1,i2-1,a2,i4,i5-1)
     and SameSet(a1,i2,i3,a2,i5,i6) )
  imply SameSet(a1,i1,i3,a2,i4,i6) == TRUE;
  など


prim8:
(* SameArray なら SameSet *)
  ( ArrayBottom ≦ i1 ≦ i2 ≦ ArrayTop
     and SameArray(a1,i1,i2,a2,i1,i2) )
  imply SameSet(a1,i1,i2,a2,i1,i2) == TRUE;

など
```

図7 レベル 1−2 間の検証で用いた基本関数・述語の補題（一部）

Fig. 7 Theorems for primitives functions/predicates used at the proof between level 1 and level 2.

実際の証明作業では，どのような不変表明や基本関数・述語に関する補題，および補題の変数への代入値を用いて証明を行うかを，まず 9 時間程度かけて事前に考案し，続いて実際に本検証支援系を用いて証明を行った．そのとき，事前に考案した不変表明中の条件の抜け，基本関数・述語に関するインスタンスの不足，および代入値の誤り等によるパスの証明失敗を十数回繰り返し，証明を成功させるまでに 3 時間程度かかった．全作業時間の内訳を **表 1** に示す．

なお，検証作業にかかる手間は，プログラムの設計者と検証者が同じかどうか（異なれば，検証者がプログラムを理解するのに必要な時間が，余分にかかる），検証者の経験，および証明作業の進め方などに，かなり依存する．

この実験では設計者と検証者は同じであるが，検証者の経験は，過去に本支援系を用いて数個のごく簡単な例題の検証をしたことがある，という程度である．このことと，今回は多くの基本関数・述語の補題を証明で用いなければならなかったことから，どのような基本関数・述語の補題を導入して証明を行うかを事前に考案するのに，かなり時間がかかった．より経験を積んだ検証者であれば，表 1 に示したよりも短い時間で，証明の道筋の考案を行えると思われる．

表1 レベル1–2間の検証の手間 (SONY NEWS-5000使用)
Table 1 Working load for the proof between level 1 and level 2.

	性質SORT1	性質SORT2	停止性
証明の道筋（用いる不変表明や補題，および補題の変数への代入値）の事前の考案	約7時間	約1.5時間	約5分
証明失敗の回数	9回	4回	なし
証明失敗時の原因調査と証明の道筋の再考案	約2時間	約1時間	なし
不変表明入力などのため支援系を操作した時間（証明失敗時を含む）	約20分	約10分	約5分
検証支援系が費やしたCPU時間（証明失敗時を含む）	510秒	114秒	0.4秒

また，今回の実験では，検証者が選択できるうち，より難しいと思われる方法を用いても検証が行えることを示すため，3.3.2項のStep (b)で項書き換えを行うことにした．このため，基本関数の補題の変数への代入値の考案にも，やや時間がかかった．

しかし，不変表明や補題などを入力するために検証者が本支援系を操作した時間，および本支援系が証明（3.3.2項のStep (a)～(f)の実行）に要したCPU時間は，証明失敗時を含めても，それぞれ約30分，約10分とわずかであった．

これは，3.4節で述べたように，本支援系ではGUIなどで不変表明の修正や基本関数・述語の補題の追加，および補題の変数への代入が容易にできるようになっており，かつ，本支援系がプレスブルガー文の真偽を高速に判定できるためである．特に，3.3.2項のStep (c)で，基本関数・述語の補題のインスタンス（SORT1の証明では合計40個程度）をすべてのパスで共通に用いることにしたので，真偽を判定した各プレスブルガー文の長さ（演算子や変数などの総出現回数）が，1300程度（SORT1の場合．SORT2では500程度）とかなり大きくなった．しかし，それでも高々十数秒程度で高速に真偽判定を行えた．**表2**に，SORT1の証明が成功したときの，各プレスブルガー文の長さと（真であることの）判定時間を示す（SONY NEWS-5000使用, 100MIPS, 64MBメモリ）．なお，証明に失敗したときは，いずれのパスについても0.01～0.1秒程度で（偽であることを）判定することができた．また，SORT2の証明では0.1秒程度で各プレスブルガー文が真であることを判定することができた．

表2 SORT1の証明でプレスブルガー文の真偽判定にかかったCPU時間 (SONY NEWS-5000使用, 100MIPS, 64MBメモリ)
Table 2 Length of Presburger sentences at the proof of 'SORT1' and CPU time needed to determine the truth of the sentences.

パス	プレスブルガー文の長さ	CPU時間（秒）
S0 → S1	1255	0.83
S1 → S2	1302	1.81
S2−(setMax) → S2	1328	7.93
S2−(unsetMax) → S2	1329	12.28
S2 → S1	1433	5.60
S1 → S3	1256	2.06

表3 レベル2–3間の検証の手間 (SONY NEWS-5000使用)
Table 3 Working load for the proof between level 2 and level 3.

証明の道筋の事前の考案	約30分
証明失敗の原因調査と証明の道筋の再考案	約30分
支援系を操作した時間	約10分
検証支援系が費やしたCPU時間	13秒

停止性については，今回は繰り返し実行が二重にネストしているので，以下の二つを示すことにした：(i) S2より遷移setMaxまたは遷移unsetMaxを実行してS2に戻るループについては，関数$bound(s) - arypnt(s)$の値が，(ii) S1より遷移initMax，遷移exchangeを経てS1に戻るループについては，関数$bound(s) - LowerBound(s) - 1$の値が，それぞれループ実行中は0以上で，かつループを1回実行する度に減少すること．これらの証明に要した手間は，合計で数分程度と，ごくわずかであった（表1参照）．

レベル2–3間の検証では，検証者は**図8**に示すような基本関数・述語に関する補題を導入した．展開に用いた系列に閉路がないため，不変表明は与える必要がなかった（停止性も特に証明していない）．この場合も，本支援系を用い，レベル1–2間と同様の手順で証明を行うことができた．検証作業にかかった手間を**表3**に示す．

4.2 考　察

ここでは，4.1節の検証実験の結果に基づき，本稿で述べた仕様記述・証明方法の利点と問題点についてまとめる．

本手法の利点としては，(1) 証明の手順を考案しなくてもよいこと，(2) 一回の証明の試行を，本検証支援系がほとんど自動で行うこと，(3) 本支援系が高速

```
(* 配列のある箇所にデータを書いた後,
   他の場所を読むと元のデータが読める *)

  ( ArrayBottom ≦ i1 ≦ ArrayTop
      and ArrayBottom ≦ i2 ≦ ArrayTop
      and not i1 = i2 )
  imply iget(iput(a1,i1,i3),i2) = iget(a1,i2) == TRUE;

(* 配列のある箇所にデータを書いた後,
   同じ場所を読むと, 書いたデータが読める *)

  ( ArrayBottom ≦ i1 = i2 ≦ ArrayTop )
  imply iget(iput(a1,i1,i3),i2) = i3 == TRUE;

(* iput した部分以外は不変 *)

  ( ArrayBottom ≦ i1 ≦ i2 ≦ ArrayTop
      and ArrayBottom ≦ i3 ≦ i4 < i5 ≦ ArrayTop
      and SameArray(a1,i1,i2,a2,i3,i4) )
  imply SameArray(a1,i1,i2,iput(a2,i5,i6),i3,i4) == TRUE;

など
```

図 8 レベル 2−3 間の検証で用いた基本関数・述語に関する性質（一部）
Fig. 8 Theorems for primitives functions/predicates used at the proof between level 2 and level 3.

であること，および (4) GUI などにより，検証者が本支援系の操作を簡単に行えること，が挙げられる．

4.1 節の実験では，これらの利点を確認することができた．一回の証明の試行に支援系が費やした CPU 時間は平均で 1 分弱であり，検証支援系が高速であることを確認できた．また，支援系の操作に要した時間も，証明失敗による不変表明の修正等を十数回繰り返したにも関わらず，合計で約 30 分とわずかであり，支援系の操作性が良いことも確認できた．

逆に本手法の問題点としては，(i) 証明で真偽を判定するプレスブルガー文が大きくなることと，(ii) 証明で用いる基本関数・述語の補題，およびその変数への代入の考案に時間がかかることが挙げられる．いずれも，その原因は，2.1.2 項で述べたスタイル制限のもとで仕様を記述するために，抽象度の高い基本述語・関数（*Ordered* など）を多く導入しなければならず，このために証明でも基本述語・関数に関する補題（の変数に定数を代入して得られるインスタンス）が多く必要となるからである．

検証実験では，実際に，使用した基本関数の補題のインスタンスの数が数十個と多くなった．このため，プレスブルガー文が（長さが 1300 程度と）かなり大きくなった．しかし，本支援系は，プレスブルガー文の真偽を十数秒程度で高速に判定でき，(i) のプレスブルガー文の大きさは，実際上はそれほど問題とはならなかった．

また，多くの補題のインスタンスが必要となったことから，実際に (ii) のとおり，補題および補題の変数への代入値の考案に時間がかかった．しかし，それにも関わらず，過去にクイックソートプログラム[3]や酒屋在庫管理プログラム[4]などの証明を行ってきた経験から，代数的手法を用いた一般的な証明方法（項書き換えや場合分けなどを検証者が対話的に実行していく，従来の我々の手法）と比べ，証明作業全体に要した期間は，かなり短かったと考えられる．

検証には十数時間かかったが，形式的手法を用いるかどうかによらず，プログラムの正しさの証明では，ある程度の手間がかかることは，やむを得ないと思われる．仮に本論文で用いた例題の証明を，自然語を用いて，4.1 節で行ったのと同程度の厳密さで（各分岐条件が正しいことやプログラムが停止することの証明などを含めて）行えば，数時間から数日程度はかかると思われる．

証明において形式的手法を用いることの利点としては,「この仮定と，この推論規則から，この結論が得られる」という証明の各ステップが正しいこと（各推論で，その前提条件が確かに成り立つことなど）の確認を，機械的に誤りなく行えることが挙げられる．自然語を用いた証明では，特にステップ数の多い複雑な証明をした場合，チェックもれや誤りが入る可能性が高くなるが，本論文で提案した手法を用いることで，証明を誤りなく厳密に，かつ比較的短時間で行えることが期待できる．

5. あとがき

本論文では，順序機械型プログラムを対象とし，仕様の記述スタイルを制限したうえで，検証自動化を目指した証明手順を提案し，実際にその方法に基づく検証支援系を作成して検証実験を行った．本支援系は C 言語で作成されており，そのプログラムサイズは約 50,000 行である．

本稿で用いた例題を含め，酒屋在庫管理プログラムの検証実験[8]や同期式順序回路の検証実験[9]を通して，本論文で提案する証明手順により，我々の従来手法よりは短期間で，かつ実用的な時間で証明を行えるとの見通しを得ている．

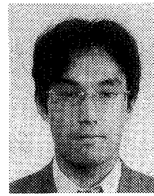
今後の課題の一つとしては，証明作業の支援の工夫等が挙げられる．そのための具体的な方法としては，証明が失敗したときに，式の構造が限られていることを利用して，どの部分が失敗の原因か指摘する機能の追加等が考えられる．

参 考 文 献

1) 大蘆雅弘，杉山裕二，谷口健一：代数的言語 ASL における抽象的順序機械型プログラムとその処理

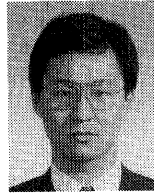
系，信学論 (DI)，Vol.J73-D-I, No.12, pp.971–978 (1990).
2) 嵩 忠雄，谷口健一，杉山裕二，関 浩之：代数的言語 ASL／＊－意味定義を中心に—，信学論 (D)，Vol.J69-D, No.7, pp.1066–1074 (1986).
3) 東野輝夫，関 浩之，谷口健一：代数的仕様から関数型プログラムの導出とその実行，情報処理，Vol.29, No.8, pp.881–896 (1988).
4) 岡野浩三，北道淳司，東野輝夫，谷口健一：代数的言語 ASL で記述した在庫管理プログラムとその正しさの証明，信学技報，SS90-29 (1990).
5) Cousot, P.: Methods and Logics for Proving Programs, Ch. 15, *Handbook of Theoretical Computer Science Vol.B –Formal Models and Semantics,* ed. Leeuwen, J.V., Elsevier Sci.Pub. (1990).
6) Cooper, D.C.: Theorem Proving in Arithmetic without Multiplication, *Machine Intelligence*, No.7, pp.91–99 (1972).
7) 森岡澄夫，東野輝夫，谷口健一：全ての変数が存在記号で束縛された冠頭標準形プレスブルガー文の真偽判定プログラム，信学技報，SS95-18, pp.63–70 (1995).
8) 森岡澄夫，岡野浩三，東野輝夫，谷口健一：関係データベースを用いた在庫管理プログラムの記述とその詳細化の正しさの証明，情報処理学会論文誌，Vol.36, No.5, pp.1091–1103 (1995).
9) Kitamichi, J., Morioka, S., Higashino, T. and Taniguchi, K.: Automatic Correctness Proof of the Implementation of Synchronous Sequential Circuits Using an Algebraic Approach, (Kropf, T. and Kumar, R. eds.), Vol.901 of *Lecture Notes in Computer Science*, pp.165–184, Springer-Verlag (1995).

(平成 6 年 4 月 11 日受付)
(平成 7 年 7 月 7 日採録)



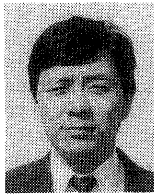
森岡 澄夫（学生会員）

昭和 43 年生．平成 4 年大阪大学基礎工学部情報工学科卒業．平成 6 年同大学院博士前期課程修了．現在，同大学院博士後期課程在学中．代数的手法を用いたソフトウェア・ハードウェアの形式的検証などに関する研究に従事．平成 6 年度より日本学術振興会特別研究員．日本ソフトウェア科学会会員．



北道 淳司（正会員）

昭和 40 年生．昭和 63 年大阪大学基礎工学部情報工学科卒業．平成 3 年同大学院博士後期課程中退．同年同大学基礎工学部情報工学科助手，現在に至る．ハードウェアの仕様記述と設計などに関する研究に従事．電子情報通信学会会員．



東野 輝夫（正会員）

昭和 31 年生．昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学院博士課程修了．工学博士．同年同大学助手．平成 2 年，6 年モントリオール大学客員研究員．平成 3 年大阪大学基礎工学部情報工学科助教授，現在に至る．分散システム，通信プロトコル等の研究に従事．電子情報通信学会，IEEE_CS，ACM 各会員．

谷口 健一（正会員）

昭和 17 年生．昭和 40 年大阪大学工学部電子工学科卒業．昭和 45 年同大学院基礎工学研究科博士課程修了．工学博士．同年同大学基礎工学部助手．現在，同情報工学科教授．計算理論，ソフトウェアやハードウェアの仕様記述・実現・検証の代数的手法および支援システム，関数型言語の処理系，分散システムや通信プロトコルの設計・検証法などに関する研究に従事．