GDB とシステムモデルを用いたソースコード検証器の開発

永 藤 直 行[†,††]

一般的に，ソフトウェアのライフサイクルには 2 つの検証工程がある．1 つはシステムの機能的な仕様が記述されたモデルの検証であり，もう 1 つはそのシステムの実際に動作する C などのようなプログラミング言語により記述されたコード（ソースコード）の試験である．我々は，この両方の工程で利用可能な検証器を開発した．モデル検証は，質の高い試験を行う助けとなる．すでにいくつかの検証ツールが存在するが，それらはモデルかソースコードかのいずれかを対象としている．我々のツールはその両方を検証，試験することが可能である．ソースコードを検証するときには，GDB 上で試験対象となるコードを実行し，そのコードに対応するモデルの構成要素を除いたモデルの一部と同期させて試験する．モデルは，我々のツールではプロセス代数に基づいた言語により記述される．ソースコードの状態空間は削除された構成要素と他の要素の間の通信に利用されるアクションに対応したソースコード上のブレイクポイントの集合として定義される．このツールでは最初の検証工程で作成されたシステムのモデルをソースコードの試験に再利用することが可能である．本稿では，簡単な例として ECHO サービスを示す．

Direct Model Checking of Real Codes Using GDB and System Models

Naoyuki Nagatou[†,††]

In general, there are two checking processes within the lifecycle of a system. One is verification of models describing functional specification, and the other is testing of real codes implementing the system. We developed a model checking tool can being used on both processes. Model checking systematically explores the state space of systems. The systems are expressed as the model or real code (or source code) written in programming languages such as C. There are several model checking tools that handle either the models or the real codes, but not both. In testing processes of the real codes, our model checking tool checks both the models and the real codes. The tool executes the binary code on GDB, then examines a composition with the model from which the portions of the model that correspond to the code being executed have been eliminated. A state space of the real code is a set of GDB's breakpoints corresponding to actions to communicate between the eliminated portion and others, so the model is written in a modeling language based on process algebra. This tool enables the re-use of the same model that was used in verification processes. We will illustrate ECHO service as a simple example.

1. はじめに

システムはより複雑になっているにもかかわらず，試験のための時間は限られている．この限られた時間で質の高い試験を行うことはむずかしい．モデル検証は，この問題を解決する助けとなる．モデル検証器は，機能的な振舞いを定義したシステムの抽象的なモデルが与えられ，そのモデルの可能な状態の集合を生成し，各状態である性質が真であるか偽であるかを検査する．システムの機能的な振舞いを定義したものがモデルであり，性質は，そのモデルにのぞまれる特徴を表している．一般的に，システムの試験には 2 つの検査工程があり，1 つはシステムの機能的な仕様が記述されたモデルの検証で，もう 1 つはそのシステムの実際に動作するコード（ソースコード）の試験である．後者は，C などのようなプログラミング言語により記述さたコードのデバックである．この 2 つの工程それぞれで検証器が開発されている．SPIN[5] のような従来の検証器ではモデルだけを扱い☆，最近の検証

† 東京工業大学大学院情報理工学研究科計算工学専攻
Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
†† 有限会社プレシステム
PRESYSTEMS, Inc.

☆ SPIN バージョン 4.0 以降では，C のコードを Promela のコードに埋め込むことが可能である．

器[4),9),11)] ではソースコードだけを扱う．また，Java PathFinder[13)] は Java のソースコードを SPIN のモデル記述言語である Promela[5)] に変換してソースコードを間接的に試験する．CMC[9)] は，ソースコードの試験では試験環境をユーザが準備しなければならない．しかし，その環境自体の正当性を保証することはむずかしい．我々は，モデル検証を行った後，ソースコードを直接試験するモデル検証器を開発した．システムモデルをそのコードの試験環境として利用する点に特徴がある．システムモデルはコードの試験の前に検証されているので，コード試験のときには，ある性質を満たすことを検証するうえで，その環境が正しいことが保証されている．ソースコードを試験するときには，我々のツールでは GDB 上で試験対象となるコードを実行する．モデルは Milner の CCS[8)] に基づいた言語により記述されており，そのソースコードに対応するモデルの構成要素を除いたモデルの一部を解釈実行し，必要なときには GDB と同期をとる．必要なときとは，ソースコードに対応する構成要素とそれを除いた残りのモデルの間の通信を表すアクションが実行されたときである．このとき，モデルとソースコードの両方に出現する変数の値を強制的に同じ値に変更する．ソースコードの状態空間は，この通信に対応した行に設定されたブレイクポイントにおける変数の値の集合として定義される．モデルの状態とこの状態の組で表されるシステム全体の状態集合を考え，各要素についてある性質を満たしているか否かを検査する．いい換えると，モデルだけの検査のときに用いた性質が，モデルの一部を実際のコードに置き換えても満たしていることを検査している．もちろん，ブレイクポイントから他のブレイクポイントの間に複数の制御経路が存在することも考えられるが，プログラムはシステムモデル以外とのインタラクションは存在せず，決定的であると仮定すると，このとき，ある値に対して一意の結果を返す．どの経路を実行するかはモデルから与えられた値によって制御され，その値自体は，適当な値がユーザなどにより与えられ検証中は不変である．したがって，我々の検証器ではいずれか 1 つの経路を検査する．最終的な目的は，もっと一般的なシステムの検証であるが，本稿では簡単な例として ECHO サービスを示す．作成する状態の数などの性能に関する定量的な評価はまだ行っていないので，今後，行いたいと思う．

本稿の構成を以下に述べる．2 章では，関連研究について述べる．3 章では，我々の検証ツールの構成と GDB と検証器が同期をとる方法に付いて述べ，ソースコードを直接検証する手順を述べる．4 章では，この検証器が状態空間を探索するアルゴリズムについて述べる．5 章では，今回開発した検証器の制約について議論する．6 章でまとめを述べる．

2. 関 連 研 究

本研究の目的は，モデリング言語により抽象化したものを検証するのではなく，直接ソースコードを検証する検証器を開発することである．状態空間を探索する方法は，SPIN[5)] でも利用されているよく知られている方法を利用している．

2.1 従来の検証器

モデル検証は，システムの状態空間をシステマティックに検査する．そのためのツールである検証器[3),5)] は，従来からプロトコルの検証などに利用され，重要なバグを発見するために利用されてきた[7)]．SPIN のような従来の検証器は，特別なモデリング言語により記述された抽象的なモデルを検証する．この問題点は，実際に動作するコードを検証していないところにある．我々の検証器は，実際に動作するコードを直接検証する．しかし，SPIN 4.0 以降ではソースコードの断片を Promela コード中に埋め込むことが可能である．我々の検証器は，このコード断片の作りかた，考えかたが異なる．我々の検証器は，ソースコードとバイナリコードを GDB 上で実行し直接検証する仕組みを提供している．GDB を経由して複数のブレイクポイントを設定することで，SPIN のコード断片に相当するものを我々の検証器内に作りだしている．それぞれの検証器が試験のときに利用するバイナリコードについて考えると，我々の検証器が利用するバイナリコードは実際に運用する際のコードにより近いコードを利用している．

2.2 最近の検証器

近年の検証器は，ソースコードレベルで検証を行うというアイデアが利用されている．しかし，これらはソースコードだけを対象にしており，モデル検証は 2.1 節であげたツールを利用しなければならない．我々の検証器はいずれのレベルでも検証を行うことができる．以下では，ソースレベル検証を行う各検証器と個別に比較する．

Verisoft[4)] は，このようなアプローチの最初の検証器である．並行実行されるコード間の通信に関係する処理だけを観測可能であるとしている．たとえば，共有変数，セマフォ，FIFO のようなオブジェクトへの演算だけを観測するものとし，このような演算間に存在する内部演算は観測しないものとして，それら内部

演算は必ず有限であると仮定している．観測可能な演算だけによる状態遷移により構成される状態空間を探索し，それらコード間のデッドロックあるいはユーザにより定義された性質をコード内に埋め込みその性質に違反するか否かを検査する．我々の検証器も同様に観測可能な演算による状態遷移だけから構成される状態空間を検査している．CMC[9] もまた同様にソースコードレベルで試験する検証器である．これは，OS とのインタフェースや通信回線あるいはそれらのデータの状態といった試験対象であるコードの実行環境をユーザが作成しなければならない．このとき，この環境自体が正しく動作するどうかを保証していない．我々の検証器では，試験対象コードに対応するモデルの構成要素を除いた部分がここでいう環境の役割をし，このモデル自体はソースコードの試験の前に検証されているとすれば環境そのものが正しいことを保証している．また，上記 2 つの検証器は環境の非決定的な振舞いを模倣するために適当な範囲内の 1 つの整数を選択する特別な演算子を用意しているが，我々の検証器ではモデル自体に非決定的な振舞いを定義することが可能である．

Java PathFinder[13] は，Java プログラムを検証する．ソースコードを SPIN のモデル記述言語である Promela に変換し，それを検証する．ソースコードを対象としているが直接それを検証していない．我々の検証器は GDB 上でコードを実行し，直接コードを試験している．GDB が利用できるデバック情報を付加したコードを生成するコンパイラがあれば特に言語を選ばない．GNU ツールは多くの環境に移植されており，この制限は問題にならないと思われる．

これまでの検証器は，それぞれ独自のスケジューリング戦略をもちいており，それらは実際にコードが実行されるときのスケジューリングと異なるという理由から，Nakagawa ら[11] は，GDB と SBUML を利用し実際にコードが動作するときと同じスケジューリング戦略のもとで試験する検証器を開発した．状態数を抑制するためには有効なアイデアであると考えられる．我々の検証器は，システムの各構成要素のインタリーブを考え，それらすべてを検査するわけではないが，実際のスケジューリングと等価な状態を必ず検査する．一方，彼らの検証器では複数の GDB を実行し複数のコードを同時に検証することが可能である．我々の検証器では同時に複数のコードを実行することはできない．実行しているコードに対応する構成要素を除いたモデルがその代わりをする．

以上，いくつかのソースコードレベル検証器について述べてきたが，システムモデルをそのコードの環境として利用する点に特徴がある．システムモデルはコードの試験の前に検証されているのでコード試験のときには，ある性質を満たすことを検証するうえでは，その環境の正しさが保証されている．

3. 検証器の設計

この章では，我々の検証器の概要について述べる．はじめにこの検証器の概略を述べ，次にこの検証器のためのモデル記述言語で記述されたモデルと線形時間論理の式の状態について述べる．そのとき，簡単な例として ECHO サービスのモデルとそのサーバのソースコードを与え，最後に，ソースコードを直接検証する様子を説明する．

3.1 検証器の概要

我々の検証器は 3 つのサブシステムから構成される（**図 1** 参照）．1 つはモデルを解釈実行する検証器本体であり，2 つ目は GDB を制御するサブシステムであり，3 つ目は線形時間論理式をオートマトンに変換し，さらにそれを，1 つ目の検証器本体が解釈実行することができる式の集合に変換するためのサブシステムである．それぞれ，Verifier，GDB interface，LTL2CCS と呼ぶことにする．最初，ユーザによりモデルと性質が与えられる．ソースコードを試験するときには，さらにソースコードとそのバイナリコードが与えられる．LTL2CCS は，与えられた線形時間論理式をオートマトンに変換し，それをさらに Verifier が解釈できる式に変換する．この式は，CCS[8] に基づいた言語により表現されるプロセス式の集合として表される（プロセス式については 3.2 節で述べる）．Verifier はその式とモデルを表す式（モデルも同様にプロセス式の集合として表される）を合成し，初期状態から可能な遷移により生じさせられる状態を生成していき，そのオートマトンによって受理される遷移の列が存在するか否かを検査する（この詳細は 4 章で述べる）．状態を生成しているとき，もしある状態で 2 つの状態遷移が可能なときには一方を実行した後，バックトラックしてもう一方の状態遷移を実行する（プロセス式の性質上，1 つの状態からは多くとも 2 つの部分プロセス式の状態遷移を調べるだけでよい）．GDB interface は，ソースコードを試験するときだけ利用される．このとき，最初に一度だけ，GDB とのパイプを生成し，試験対象となるコードファイル名を指定して GDB を子プロセスとして MI（Machine Interface）モードで起動する．以後，バイナリコードの制御は，Verifier の要求に応じて GDB interface がデバッカを通して行

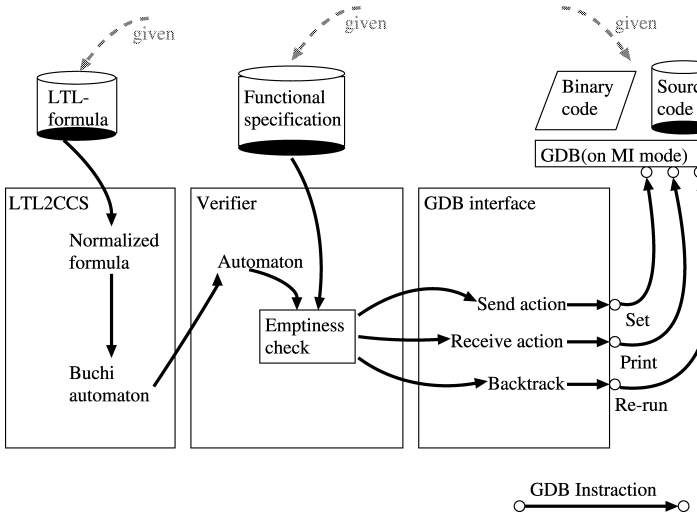

図 1 検証器の構成
Fig. 1 The architecture of our model checking tool.

う．バイナリコードの現在の状態はデバッカが保持しており Verifier が必要とする変数の値をデバッカから取得したり，デバッカが保持している変数の値を変更したりする．ある変数の値を取得する必要があるとき，すなわち受信アクションを Verifer が解釈するときには，GDB の print コマンドを利用し，バイナリコード中の変数の値を変更する必要があるとき，すなわち送信アクションを解釈するときには set コマンドを利用する．このようなアクションは，バイナリコードとモデルの同期を必要とするアクションだけに限られる．同期を必要とするアクションは，ユーザによりモデル内で bind 演算子を用いて指定される（bind 演算子については 3.3 節で詳しく述べる）．GDB interface は，バイナリコードとモデル間の実行された同期アクションの履歴を保持する．Verifier は，合成した式，つまり状態空間を表すグラフ上を深さ優先で探索するので，バックトラックがかかったときにはバイナリコードの状態を戻す必要がある．初期状態からその子供のノードを順に探索し，あるノードの一方の子供を探索したあとでバックトラックが生じる．その子供への遷移がモデルとコードとの同期を必要とするアクションであるときには，GDB interface が保持している履歴に含まれる最後の状態の 1 つ前の状態まで再実行することを要求する．この要求を受け取った GDB interface は GDB の再実行コマンド re-run を利用し，バイナリコードを再実行する．ここまで，3 つの構成要素の連携のしかたについて述べた．次に，Verifier がモデル記述言語を解釈実行する様子について述べる．

我々は，モデルの記述体系としてプロセス代数に基づいたモデル記述言語を提案している．この言語の処理系である Verifier について述べる．この記述体系では，システムは複数の構成要素からなるものとしてシステムをモデル化し，それぞれの構成要素は逐次処理を行い，並行に実行される．このモデル記述言語には，送信アクションと受信アクションと呼ばれる特別な式の要素があり，このアクションを実行することによりシステムの状態が遷移する．Verifier は，基本的にはインタプリタと同様な機構を持っていて，この言語を解釈実行する．並行性を実現するためにこのインタプリタは内部に継続の機構を持ち，コルーチンの概念を利用して並行性を実現している．現在処理している部分プロセス式の部分をホールとした継続を保持し，その部分式の実行可能なアクションのうちいずれか 1 つを実行したあとの部分式と継続からあらたにプロセス式を構成し，処理を継続する．各構成要素間の通信はスタックバッファを通して行われる．スタックバッファは，インタプリタの環境のことであり，送信アクションが実行されたときには，アクション名にその送信アクションが送信する値が対応づけられていると考えて，環境にこのアクション名と値の組を保存する（**図 2** 参照）．受信アクションが実行されたときには，アクション名と同じアクション名を持つ組を環境からみつけてきて，その組の値を受信アクションの変数に対応づける．環境から最初にみつかった組を利用するので擬似的に各構成要素間の通信路はスタックバッファのようになる（**図 3** 参照）．

ソースコードはシステムの構成要素の 1 つをモデルから実際に実行するコードに置き換えたものである

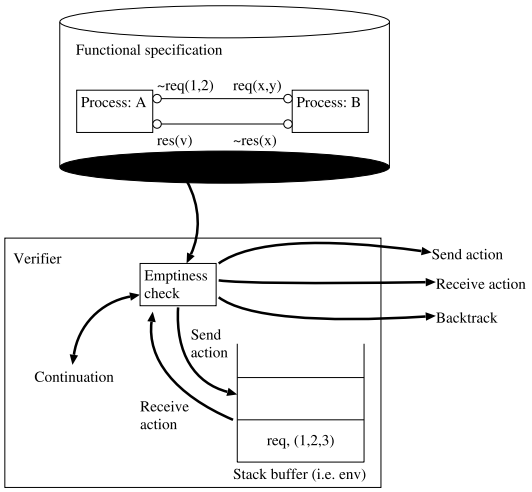

図 2 検証器の概要
Fig. 2 Outline of Verifier.

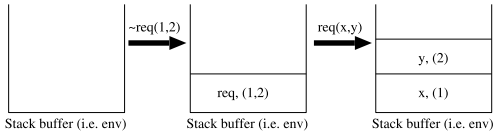

図 3 通信路の様子
Fig. 3 An aspect of channels.

と考える．そのコードの状態遷移にはモデルから観測可能な演算によるものとそうでないものがある．ソースコードに置き換えられた構成要素と他の構成要素との間の通信を表しているアクションに対応する演算はモデルから観測可能であるとし，それ以外の演算はモデルから観測不可能であるとする．アクションとソースコード中の演算の対応はユーザによって `bind` 演算子を利用して指定される（詳細は 3.3 節で述べる）．Verifier は，観測可能な演算を実行した直後のプログラムの状態だけを生成していく．このとき，観測不可能な演算の列による状態遷移の列は必ず有限でなければならないとする．もし，観測可能な遷移の間に観測不可能な演算の無限列が存在するときにはソースコードの試験はそれ以上進まない．以上のように，我々の検証器はソースコードの試験では，モデルから観測可能な演算を実行した直後の状態だけからなる状態空間を検査する．この章の残りの部分では，簡単な例をもちいてソースコードを試験する過程を説明する．

3.2 モデルと性質の状態集合

Verifier のレベルでは，モデルも性質も同じ言語によりプロセス式として表現される．最初に，その記述言語について述べる．P は，プロセスの名前の集合 P の要素，A はアクションの名前の集合で $a, b, c \cdots \in$ A とし，C は，整数または文字列の集合上の変数または定数の関係式の集合 C の要素とし，x は，整数または文字列の変数の集合 V の要素とし，e は，整数または文字列の集合 D 上の値，そして E はプロセス式の集合として，$E \in$ E とする．このとき，E は以下のように定義される．

$$E ::= P(x, \cdots) \mid E + +E \mid E||E \mid$$
$$\texttt{if}\ (C)\ (E)\ (E) \mid$$
$$\tilde{}a(e_1, \cdots, e_n) : E \mid a(x_1, \cdots, x_n) : E.$$

それぞれ，プロセス定数，プロセス和，プロセス合成，条件といい，残りの 2 つをアクション前置ということにする．アクション前置には送信アクションと受信アクションの 2 種類がある．プロセス定数は特別な演算子 `define` により定義され，プロセス和はいずれかのプロセスを非決定的に選択し，プロセス合成は 2 つのプロセスの同期を表し，条件は C の値によりプロセスを選択実行する．送信アクションはいつでも実行可能であり，それが実行されたときにはその値 $e_1, \cdots, e_n$ がアクション名とともにスタックバッファにプッシュされる．受信アクションは同じ名前を持つ送信アクションが実行されたあとであれば，いつでも実行可能であり，それが実行されたときにはスタックバッファからその名前と値 $e_1, \cdots, e_n$ がポップされ，それぞれの変数 $x, \cdots$ にこの順番でバインドされる．**図 4** は ECHO サービスのモデルの例である．2 行目から 26 行目まではクライアントの振舞いを表しており，28 行目から 29 行目はサーバの振舞いを表している．

性質は線形時間論理の式としてユーザから与えられる．ただし，その式に Next 演算子を含んではならない．この検証器では，原子論理式は集合

V ∪ D ∪ {`true`, `false`}

上の関係 <=，<，>=，>，==，！をもちいて表される．それぞれの意味は C 言語のそれと同じである．LTL2CCS は，この線形時間論理の式をオートマトンに変換し，そのオートマトンを条件，アクション前置，`accept` という特別なアクション，さらに `Abort` という遷移が存在しないことを表す特別なプロセス定数と状態を表す適当なプロセス定数をもちいて表現する（**図 5** 参照）．具体的には，各状態を表すプロセス式は，

```
(define State_id ()
  (if (atomic_propositions)
      (accept:State_id)
      (Abort_or_State_id)))
```

の形をしていなければならない．`State_id` は状態名を表す適当な識別子であり，(`accept` : E) が受理状態を表す部分プロセス式となる．条件の

```
;(bind srv_res "target:echos:10")
;; Timer
(define Timer()
  (start(init_v):Timer1(init_v)))
(define Timer1(time)
  (~tick(time-1):tick(time):
    (if (time=0)
        (~timeout(TRUE):STOP)
        (Timer1(time)))))
(define Timeout()
  (timeout(zz):
    (if (zz)
        (~display("TIMEOUT"):~srv("quit"):STOP)
        (stop))))
;; Echo client
(define Recv1()
  (srv_res(yyy):
    (Send("TEST",n-1))))
(define Recv() (Recv1++Timeout))
(define Send(x,n)
  (if (n=0)
      (STOP)
      ((~srv(x):~start(5):
        (Recv)))))
(define Echo_client()
  (Send(100,5)||Timer))
;; Echo server
(define Echo_server()
  (srv(req):~srv_res(req):STOP))
;; Initial process
(Echo_server || Echo_client)
;(Echo_client)
```

図 4 ECHO サービスの例
Fig. 4 A simple ECHO service.

atomic_propositions は状態論理式で原子論理式の集合であり，Abort_or_State_id はプロセス式である．条件の then 部と else 部の部分プロセス式は逆でもよい．このような形のプロセス式の集合として性質を表す．与えられた式はモデル検証時と同じものがソースコード試験時にも利用される．

これまで述べたようにモデルも性質もプロセス式の集合として定義される．モデルは，有限個のプロセス式の閉じた集合として定義されていなければならない．閉じているとは，各プロセス式に出現するプロセス定数がすべて，その集合の要素でなければならないということである．よって，モデルは

```
(define Node0002 ()
  (if (! ((x) = (y)))
      (accept: Node0002)
      (Abort)))
(define Init ()
  (if (! ((x) = (y)))
      (accept: Node0002)
      (Init)))
(define _MC_START () (Init))
```

図 5 線形時間論理式の記述
Fig. 5 A description of the LTL formula <> ! (x=y).

$$\{(\mathsf{define}\ P_i\ (x,\cdots)\ E_i):\ 1 \le i \le n\}$$

となる．また，モデルには必ず 1 つだけ初期プロセス（P_i）が存在しなければならない．ここで，$1 \le i \le n$ である．すなわち，モデル M として

$$M \equiv \{(\mathsf{define}\ P_i\ (x,\cdots)\ E_i):\ 1 \le i \le n\} \cup \{(P_i):\ 1 \le i \le n\}$$

となる．このとき，モデルの状態空間は，初期状態をこの初期プロセスとし，遷移関数 $eval : \mathsf{E} \to \mathsf{E}$（付録 A.1 を参照）によりその初期状態から到達可能な部分プロセス式の集合として表される．到達可能とは，ある状態から有限回の $eval$ の適用によってその状態にいたる遷移の列が存在することである．初期プロセス式 E としたとき，その部分プロセス式の集合を $Sub(E)$ として，これは以下のように定義する．このように定義されたプロセス式の集合は自然に遷移について閉じている．$Sub(E)$ が遷移について閉じているとは，すべての $E_i \in Sub(E)$，すべてのアクション a による遷移について，$E_i \xrightarrow{a} F_i$ ならば F_i もまた $Sub(E)$ の要素であるということである．

$$Sub(\tilde{}a(e_1,\cdots,e_n) : E) = \{\tilde{}a(e_1,\cdots,e_n) : E\} \cup Sub(E)$$

$$Sub(a(x_1,\cdots,x_n) : E) = \{a(x_1,\cdots,x_n) : E\} \cup Sub(E[e_i/x_i]) \quad (\text{if there is } \tilde{}a(e_1,\cdots,e_n))$$

$$Sub(E++F) = \{E++F\} \cup Sub(E) \cup Sub(F)$$

$$Sub(E||F) = \{E||F\} \cup \{E'||F' :\ E' \in Sub(E) \text{ and } F' \in Sub(F)\}$$

$$Sub(\mathtt{if}\ (C)\ (E)\ (F)) = \{\mathtt{if}\ (C)\ (E)\ (F)\} \cup Sub(E) \quad (\text{if } C = true)$$

$$Sub(\text{if}\ (C)\ (E)\ (F)) = \{\mathtt{if}\ (C)\ (E)\ (F)\} \cup Sub(F) \quad (\text{if } C = false)$$

$$Sub(P(e_1,\cdots,e_n)) = \{P(x_1,\cdots,x_n)\} \cup \{Sub(A[e_i/x_i])\}$$
(if (define $P(x,\cdots)$ A))

以降では，この集合を E と表すときもある．

3.3 ブレイクポイントの設定

プログラムの状態空間はモデルから観測可能な演算の実行直後の状態の集合として定義される．ここでは，この状態を定義する方法について述べる．

ソースコードを試験するときには，このソースコードに対応するプロセスと他のプロセスの間のアクションをソースコードの特定の行にユーザは対応づけなければならない．そのために特別な演算子 bind が用意されている．これは指定されたアクションに対応する行を指定し，その行の直後に GDB のブレイクポイント設定する．このブレイクポイントにおける変数の値の集合がプログラムの状態になる．

我々の検証器では，システムは複数のプロセスから構成され，

$$E_1||E_2||\cdots||E_n$$

の形を持っていると考える．それぞれは逐次的に実行され，各プロセスは送信アクションと受信アクションによって互いに通信または同期をとっている．送信アクションはいくつかの値 e_i を持ち，

$$\tilde{}a(e_1,\cdots,e_n)$$

の形をしており，いつでも実行可能である．受信アクションは

$$a(x_i,\cdots,x_n)$$

の形をしていて，同じアクション名を持つ送信アクションが実行されたあとであれば実行可能であり，そうでないときには実行が遅れさせられる．この 2 つのアクションはアトミックに実行される．よって，この 2 つのアトミックなアクションの実行は

$$x_i := e_i$$

のような代入文の実行と見なすことができる．一方，ソースコードにおいて，状態はいくつかの変数の値の集合として表されるとすると，このとき，その状態を変更するのは代入文であると考えられる．その代入文をアクションの組の実行に対応させる．このような V の要素に影響する状態遷移を対応づけることは自然である[6]．しかし，すべてのソースコード内の代入文を対応づけるわけではない．このことを次に述べる．

いま，E_i がソースコードに置き換えられたとし，$\{E_j : j \neq i\}$ の各プロセスとの通信を考える．もし，E_j が送信アクション $\tilde{}a(e_1,\cdots,e_n)$ を実行したのであれば，GDB 上で実行しているコードの対応づけられた代入文の左辺の変数に強制的に $e_1,\cdots,e_n$ を代入しそのコードの実行を継続する．このようにアクションと対応づけられた演算以外にはモデルの状態遷移に影響することはない．また，もし E_j が受信アクション $a(x_1,\cdots,x_n)$ を実行したのであれば，モデル中の変数 $x_1,\cdots,x_n$ の値をその代入文の右辺の値に変更して，モデルの解釈実行を継続する☆．このように観測可能な演算以外にはモデルの実行に影響することはない．機能的な仕様を決定したときには，観測不可能な遷移はモデルにおいては未定義であり，単にプロセス間の通信に関する条件が与えられているだけである．線形時間論理の式に現れるのはその通信に関する関係だけであり途中の値の条件は含まれないはずである．仮に含まれていたとしても，それはモデルの実行時には不変である．したがって，ここで，観測不可能な遷移による経路は必ず有限であり，そのソースコードはモデル以外と通信しないと仮定すると，線形時間論理式の真偽に影響するプログラム中の演算は観測可能な演算だけである．

3.4 ソースコードの検証

この節では，ソースコードの試験を行うための各段階について述べる．モデルはすでに検証が済んでいるとする．ソースコードを作成し，次に対応するプロセスをモデルから削除する．たとえば，**図 6** のような C のソースコードを作成したとする．これはサーバのコードであるので，図 4 の 28，29 行目を削除するなどし，さらに初期プロセスを 32 行目のように変更して，モデル中のサーバを無効にする．1 行目は srv_res というアクションが echos というコードの 10 行目の代入文に対応づけられていることを bind をもちいて宣言している．コメントをはずしてこの文を有効にする．ここまで，このソースコードに対応するプロセスをモデルから削除できた．さらに，ソースコードをコンパイルし実行用のバイナリコードを生成する．このとき，必ずデバック用のコードがそのバイナリコードにコンパイル時のオプションによって，埋め込まれていなければならない．なぜなら，モデルとの同期をとるとき，変数名は必ず同じ名前で，その変数の情報が利用されるからである．これらの準備が終わったあとで，もう一度，モデル，線形時間論理式，試験コード，ソースコードを検証器に与える．このとき，性質はモデルの検証で利用したものと同じものをもちいる．検証アルゴリズムについては，4 章で述べる．

CMC はコードの検証のために試験環境をユーザが

☆ ここで，複数の変数や値が出現している理由は，C 言語の構造体のような複雑な構造を表現するためである．対応させる順番は処理系に依存する．

```
 1: void TCPecho(int sock)
 2: {
        ...
 3:     for(;;){
 4:        n = read(sock,recv_data,4);
 5:        if (!strcmp("exit",buf)){
 6:           shutdown(sock,2);
 7:           close(sock);
 8:           return;
 9:        }
10:        send_data = recv_data;
11:        send(sock,send_data,outchars,0);
12:     }
13: }
14: int main(int argc,char *argv[])
15: {
        ...
16:     listen(s,BACKLOG);
17:     for(;;){
18:        int i = sizeof(sin);
19:        if ((t = accept( ... ))<0)
20:           exit(1);
21:        TCPecho(t);
22:        close(t);
23:     }
        ...
24: }
```

図 6 ECHO サーバの例（echos.c）
Fig. 6 A simple ECHO server (echos.c).

作成しなければならない．OS や外部要因をエミュレートする複雑な環境を作る必要があるとき，その試験環境自体が正しいことを保証することはむずかしい．しかし，我々の検証器では試験環境としてのモデルは，すでにモデル検証によりその正しさが保証されている．Sub(Echo_client) はソースコード試験環境の状態遷移による状態の集合と見なすことができ，与えられた性質に関して，そのモデルの検証によって正しいことが保証されている．

4. 検証アルゴリズム

モデル，線形時間論理，ソースコードが前述のように与えられたとする．我々の検証器は，システムの構成要素である各プロセス間の通信を表すアクションのいくつかの有限または無限の列を実行することで生成される状態空間を探索し，モデルと性質の両方のオートマトンが受理するアクションの列がないことを検査する．受理するとは，そのアクションの列により無限に多くの回数，受理状態に到達することである．モデルはすべての状態が受理状態であると考える．一般には，公平性によってはこのように仮定できないが，我々の検証器ではスケジューリングにより公平性に違反する状態遷移自体が存在しえないので，このように仮定する．モデルはすべての状態が受理状態であると仮定したとき，両方のオートマトンを合成したときの受理状態は線形時間論理式から変換されたオートマトンの受理状態だけに依存する．3.2 節で述べたように性質のオートマトンは (accept : E) というプロセス式が受理状態を表す．

我々の検証器は，プロセス式の集合を表す 2 つのデータ構造を持つ．それらは，状態空間を生成するときに利用される．一方は，初期プロセスから生成されるプロセス式の列を保存し，到達可能なすべての状態を検査することを保証する．これをハッシュテーブルと呼ぶことにする．もう一方は，初期状態から到達可能な終了状態からバックエッジを発見する際に利用される．これをマーク付けテーブルと呼ぶことにする．

4.1 初期状態

3.2 節で述べたようにモデルはプロセス式の集合である．モデルの各状態は部分プロセス式自体であり，状態空間は部分プロセス式の集合である．この集合には初期プロセス式が含まれており，その初期プロセス式から状態を生成する．よって，モデルの初期状態は，初期プロセス自体である．線形時間論理の式から変形したオートマトンの初期状態は（_MC_START）という特別な初期プロセス式である．試験するコードの初期状態は，そのコードのためのリンカによる初期化モジュールが実行された直後の状態である．ソースコードにおいては，C 言語の場合 main 関数の先頭の行の直前の状態に対応する．この状態は，デバッカによって保持されている．また，各変数の初期値はユーザによりあらかじめ決められているものとし，その値は検証中には変化しない．

4.2 状態の生成

On-the-fly に状態遷移グラフを作成するために，この検証器は，ある状態から可能なすべての次の状態の集合を計算できなければならない．状態空間には，いくつかの遷移が可能な状態が存在する．プロセス式で表されたモデルの各状態は，A.1 節で定義された意味から，その状態で解釈実行する演算子がプロセス合成 || とプロセス和 ++ であるときに複数の遷移が可能であり，このときそれぞれの遷移による個々の状態を生成できなければならない．これら 2 つの演算子がプロ

セスの非決定的な振舞いを表現している．このプロセス式の演算子以外の演算子は，決定的な振舞いを表しているので，単にその部分式を，逐次，遷移させていけばよい．++，|| はいずれも被演算子が 2 つであるので，一方の部分式を遷移させたあとで，もう一方の部分式を遷移させればよい．++ を解釈実行するときには，一方の部分プロセス式を評価し，その部分プロセス式から可能な部分状態集合を計算したのち，状態をもどしてもう一方の部分式を同様に遷移させる．|| についても各部分プロセス式を個別に評価するが，この場合には，一方の部分式だけがかわり，現在のプロセス式はそのままである．つまり，プロセス式 $E||F$ としたとき，E を評価して E' となったならば $E'||F$ となる．さらに E から可能なすべての状態を計算したならば，こんどは $E||F$ から F を同様に評価する．++ はいずれか一方の部分式を選択したとき，もう一方の部分式は，その後，他のプロセスに影響することはないが，|| はいずれか一方の部分式を選択したときでも，もう一方の部分式は，その後も互いに影響しあうのでこのような違いが生じる．

我々の検証器は，次の処理を繰り返して状態グラフ全体を生成していく．最初，初期プロセスからはじめ，現在の状態をハッシュテーブルに登録する．その状態から可能な遷移の 1 つによる状態を生成し，生成した状態がまだハッシュテーブルに登録されていなければ，さらにその状態からこの処理を繰り返す．もし，現在の状態が受理状態であれば，その状態をルートとして，もう一度ルートから可能な遷移による状態を生成していく．ルートをマーク付けテーブルに登録し，その状態から可能な遷移による状態を生成し，生成した状態がハッシュテーブルにすでに登録されている状態であれば，初期状態から受理状態を含む閉路へいたる経路が存在しそのアクション列は受理されるとして探索を中止する．そうでないときには，生成した状態から同様の処理を繰り返す．このように，順次，状態を生成しながら状態空間全体を探索する．これは Double DFS [2] と呼ばれる方法であり，この疑似コードを**図 7** に示す．ここで，生成した状態がそれぞれのテーブルに含まれているかを判断するために，2 つの状態が等しいという概念を定義する．状態が等しいとは，そのまま 2 つのプロセス式の字句がすべて同じであるということである☆．よって，あらたに生成したプロセス式と同じプロセス式がテーブルに含まれているかどうかを調べればよい．

☆ 我々のモデル記述言語は名前空間が 1 つだけであるので，同じ名前で異なる引数のプロセス定数やアクション前置を作成することはできない．

```
bool EmptynessCheck(model, property)
{
   dfs1(model,property);
   return(False);
}
dfs1(subexp_m, subexp_p)
{
   hash(subexp_m);
   forall( (successor_m,successor_p)
             =eval(subexp_m,subexp_p) ){
      if ( (successor_m,successor_p)
           is not in hash table)
         dfs1(successor_m,successor_p);
      if ( subexp_p is an accepting state )
         dfs2(subexp_m,subexp_p);
   }
}
dfs2(subexp_m, subexp_p)
{
   marked(subexp_m);
   forall( (successor_m,successor_p)
             =eval(subexp_m,subexp_p) ){
      if ( (successor_m,successor_p)
           is in hash table)
         exit(True);
      if ( successor_m is not marked )
         dfs2(successor_m,successor_p);
      else
         return;
   }
}
```

図 7 検証アルゴリズム
Fig. 7 Pseudocode for our model checking argorithm.

次に，ソースコードの状態の生成について述べる．プログラムの状態は 3.3 節でも述べたように設定されたブレイクポイントにおける各変数の状態の組として表される．C や C++で書かれたプログラムの振舞いは本来決定的であり，ある値に対していつも同じ制御経路を実行し，必ず一意にその結果が得られる．よって，あるブレイクポイントから，その値に対する 1 つの実行経路だけを，次のブレイクポイントまで実行する．もちろん，複数の実行経路が存在する可能性があるが，そのときには，モデル中で非決定的な振舞いとして表される．たとえば，ソースコード中の変数 $\mathtt{x} \in \{1,2,3\}$ の値によって分岐するとき，直前のブレ

イクポイントに対応するアクション a として

$a(1) : E++a(2) : E++a(3) : E$

とユーザはモデルを記述すればよい．もし，モデルがある状態までもどるときには，このプログラムはモデルがもどる状態に対応するブレイクポイントまでGDB により再実行される．これを繰り返しながらプログラムのすべての状態を生成する．

4.3 正当性の検査

この検証器では，モデル検証を行っている間，3.2 節で述べた各状態について線形時間論理式から得られたオートマトンが遷移可能であるか否かを検査する．この検証器は，状態を生成しているときの各状態での変数の値をその内部に保持しており，$env : \mathsf{V} \to \mathsf{D}$ として，モデルの各状態は env によりラベル付けされている．ある線形時間論理の原子論理式を f としたとき，3.2 節の後半で述べたように f は，プロセス式 “条件” の条件部分に出現する．この f を env のもとで評価したときの値 $evalvar(f, env)$ を検査することで線形時間論理式のオートマトンの次の遷移を判断する．

モデルの一部がプログラムに置き換えられたとき，観測可能な演算の直後，すなわちブレイクポイントでの状態を観測するだけで十分であるかを見ていく．ソースコードの試験のときには，f の中に出現する変数で検証器に保持されていない変数の値を GDB から得ることにより $evalvar(f, env)$ を検査する．あるプログラムの状態 s' とする．この状態で $evalvar(f, env)$ に影響する変数が定義されたとすると，観測不可能な演算は有限であると仮定しているので次の観測可能な演算を実行した直後の状態 s に，この定義は必ず伝搬してくる．よって，$evalvar(f, env)$ の検査に影響するソースコード内の変数の変化は，観測可能か不可能かにかかわらず必ず観測可能な代入文まで伝搬してくる．したがって，観測可能な演算の直後の状態で f を評価すればよい．

5. ソースコード検証器の制約

この章では，今回開発した検証器の制約について議論する．最初の 2 つは，ソースコード検証一般の問題であり，残りは，我々の検証器特有の問題である．

Verifier は，観測可能な演算を実行した直後のプログラムの状態だけを生成していく．このとき，観測不可能な演算の列による状態遷移の列は必ず有限でなければならないとする．もし，観測可能な遷移の間に観測不可能な演算の無限列が存在するときにはソースコードの試験はそれ以上進まない．

システムコールのようなオペレーティングシステムとの通信を行うコードは，そのときのオペレーティングシステムの状態に依存して検証結果が異なり，すべての場合を検査することはできない．しかし，モデル中にオペレーティングシステムの一部の振舞いをモデル化すれば試験することが可能である．たとえば，ライブラリ関数 malloc() を考える．モデル中にこの振舞いを

$alloc(1) : E++alloc(0) : E$

とユーザがモデルを記述すれば，malloc() の成功あるいは失敗したときのコードの振舞いを試験することができる．この例では成功と失敗というように抽象化したが，抽象化のレベルは，ユーザが適当な粒度を考えればよい．

開発の都合上，リモートデバッキングは利用できない．リモートデバッキングを行うとき，GDB では再実行ができないからである．この制約は，組み込みシステムのようにクロス開発環境では，実機上で動作するコードを検査することができないことを意味する．また，ソースコードとモデルの間では，整数型のデータだけを相互に扱うことができる．C 言語では，真偽値，文字型，ポインタ型も整数と見なされるので現在は整数型だけを扱うようにしている．

6. 結論と今後の予定

本稿では，システムモデルと GDB を用いて，ソースコードを直接試験することを目的に開発した検証器について述べた．この検証器は，上流工程で作成されたシステムモデルとそのインプリメントであるソースコードのバイナリをデバッカ（ここでは GDB）上で動作させ，モデルと同期させることでソースコードを直接検証する．モデルあるいはソースコードのいずれかを対象とするのではなく，これらの検証を一連の動作ととらえ，1 つの検証器でいずれも検証できるようにした．これにより，モデルを下流工程でも再利用することが可能になる．さらに，コードを試験する際に利用する環境（他のプロセスや OS との通信）はモデル検証において，それ自体が正しいことが保証されている．

他のソースレベル検証器のように，複数のプログラムを同時に試験する必要があるが，現在は，ソースコードは 1 つしか同時には試験できない．今後は複数のコードを同時に動作させることができるように複数の GDB を同時に制御できるようにする予定である．

ソースコード試験のとき，3.3 節で述べたように試験データは $a(1) : E++a(2) : E++\cdots$ のように

可能なデータをユーザがモデル中に記述しなければならない．ただし，すべての可能なデータについて書く必要はない．ユーザがデータを適当に抽象化すればよい．Verisoft や CMC では，このためにランダムな値を生成する関数を用意している．我々の検証器では，抽象化したあとのデータの個数分だけプロセス和演算子をもちいて場合を分ければよい．

モデル検証において重要な問題は “状態数爆発” と呼ばれる問題である．状態空間は非常に大きくなるか，あるいは無限になる．よって，メモリや時間上の制約から状態グラフ全体を検査することは不可能である．我々の検証器は，この問題を緩和するために，データ抽象化（Data Abstraction）と Partial Order Reduction を利用した．Partial Order Reduction を利用したとき，線形時間論理の next 演算子を性質記述に用いることができない．この問題は，実行時監視と組み合わせることにより解決できると考えられる．Schneider のセキュリティオートマトン[12] を拡張し，可能な範囲で予測したり，付加情報を得たりすることにより next 演算子による性質を強制することが可能であると思われる[1),10)]．

我々の最終的な目標は，あくまでも一般的なシステムの検証である．ここでは簡単な例でしか示していないので，今後はより実際的なシステムの検証を行い，この検証器の性能について定量的な評価を行いたい．

謝辞 本システムは，IPA 未踏プロジェクトの支援をうけて開発された．関係者の方々に深く感謝いたします．

参 考 文 献

1) Bauer, L., Ligatti, J. and Walker, D.: More Enforceable Security Policies, Technical Report TR-649-02, Princeton University (2002).
2) Clarke, Jr., E.M., Grumberg, O. and Peled, D.A.: *Model Checking*, MIT Press (1999).
3) Formal Systems (Europe) Ltd: Failures divergences refinement User Manual and Tutorial (1994).
4) Godefroid, P.: VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software, *Computer Aided Verification*, pp.476–479 (1997).
5) Holzmann, G.J.: *The Spin Model Checker, Primer and Reference Manual*, Addison Wesley (2003).
6) Lamport, L. and Schneider, F.B.: The “Hoare Logic” of CSP, and All That, *Programming Languages and Systems*, Vol.6, No.2, pp.281–296 (1984).
7) Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR, *Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*), pp.147–166, Springer-Verlag, Berlin, Germany (1996).
8) Milner, R.: *Communication and Concurrency*, Prentice-Hall (1989).
9) Musuvathi, M., Park, D., Chou, A., Engler, D.R. and Dill, D.L.: CMC: A Pragmatic Approach to Model Checking Real Code, *Proc. 5th Symposium on Operating Systems Design and Implementation* (2002).
10) Nagatou, N. and Watanage, T.: Run Time Detection of Covert Channels, *The 1st International Conference on Availability, Reliability and Security*, Vienna, Austria, pp.577–584, IEEE Computer Society (2006).
11) Nakagawa, Y., Potter, R., Yamamoto, M., Hagiya, M. and Kato, K.: Model Checking of Multi-Process Applications Using SBUML and GDB, *The Internatinal Confernce on Dependable Systems and Networks*, pp.215–220 (2005).
12) Schneider, F.B.: Enforceable Security Policies, *ACM Trans. Information and System Security*, Vol.3, No.1, pp.30–50 (2000).
13) Visser, W., Havelund, K., Brat, G. and Park, S.: Model Checking Programs, *15th IEEE International Conference on Automated Software Engineering*, pp.3–11 (2000).

付 録

A.1 モデル記述言語の操作的意味

各プロセス演算子の操作的意味（**図 8** 参照）を定義するために，次のようなオートマトン

$(Q, Act, \rightarrow, E, S_0)$

を仮定する．Q は状態の集合，すなわちプロセス式の集合とする．$Name$ はアクション名の集合とし，その要素を $a, b, \cdots$ で表し，$Act = Name \cup \{\tau\}$ として，その要素を $\alpha, \beta, \cdots$ で表す．ここで，τ は特別なアクションである．$\rightarrow \subseteq Q \times Act \times Q$ が与えられたとして，$(q_1, a, q_2) \in \rightarrow$ のとき $q_1 \xrightarrow{a} q_2$ と書くことにする．E は初期状態であり，単一のプロセス式である．S_0 はスタックの初期状態を表している．最初スタックは空である．スタックの要素はラベルと値の組であり，$a; v$ と書くことにする．ここで，$a \in ACT$，$v \in Val$ とする．その組のリストはスタックの状態を表す．このオートマトンの configuration を $[q, s] \in Q \times (Act \times Val)^*$ とする．また，スタックの操作関数を次のように定義する．push$(a; v, s)$ は s の先頭に $a; v$ を追加し，そのリストを返す．delete(a, s) は s の状態にあるスタッ

$$\frac{}{[\sim a(v):E,s] \overset{\sim a}{\rightarrow} [E, \mathrm{push}(a; v.s)]}\ \mathrm{Act}_1$$

$$\frac{\mathrm{member(a,s)}}{[a(x):E,s] \overset{a}{\rightarrow} [E, \mathrm{delete}(a,s)]}\ \mathrm{Act}_2$$

$$\frac{[E,s] \overset{\alpha}{\rightarrow} [E',s')]}{[E++F,s] \overset{\alpha}{\rightarrow} [E',s')]}\ \mathrm{Sum}_1$$

$$\frac{[F,s] \overset{\alpha}{\rightarrow} [F',s')]}{[E++F,s] \overset{\alpha}{\rightarrow} [F',s')]}\ \mathrm{Sum}_2$$

$$\frac{[E,s] \overset{\alpha}{\rightarrow} [E',s']}{[E||F,s] \overset{\sim\alpha}{\rightarrow} [E'||F,s']}\ \mathrm{Com}_1$$

$$\frac{[F,s] \overset{\alpha}{\rightarrow} [F',s']}{[E||F,s] \overset{\sim\alpha}{\rightarrow} [E||F',s']}\ \mathrm{Com}_2$$

$$\frac{[E,s] \overset{\sim\alpha}{\rightarrow} [E',s'] \quad [F,s'] \overset{\alpha}{\rightarrow} [F',s'']}{[E||F,s] \overset{\tau}{\rightarrow} [E'||F',s'']}\ \mathrm{Com}_3$$

$$\frac{[E,s'] \overset{\alpha}{\rightarrow} [E',s''] \quad [F,s] \overset{\sim\alpha}{\rightarrow} [F',s']}{[E||F,s] \overset{\tau}{\rightarrow} [E'||F',s'']}\ \mathrm{Com}'_3$$

$$\frac{[P,s] \overset{\alpha}{\rightarrow} [P',s'] \quad A \leftarrow P}{[A,s] \overset{\alpha}{\rightarrow} [P',s']}\ \mathrm{Con}$$

$$\frac{[E,s] \overset{\alpha}{\rightarrow} [E',s'] \quad \mathrm{eval}(B)}{[\mathrm{if}\ B\ E\ F,s] \overset{\alpha}{\rightarrow} [E',s']}\ \mathrm{If}_1$$

$$\frac{[F,s] \overset{\alpha}{\rightarrow} [F',s'] \quad \neg\mathrm{eval}(B)}{[\mathrm{if}\ B\ E\ F,s] \overset{\alpha}{\rightarrow} [F',s']}\ \mathrm{If}_2$$

$$\frac{[E,s] \overset{\alpha}{\rightarrow} [E',s'] \quad \alpha, \sim\alpha \in L}{[E[L],s] \overset{\alpha}{\rightarrow} [E'[L],s']}\ \mathrm{Res\ where}\ L = \{a, b, \cdots\}$$

$$\frac{[E,s] \overset{\alpha}{\rightarrow} [E',s'] \quad \alpha'/\alpha \in R}{[E\{R\},s] \overset{\alpha'}{\rightarrow} [E'\{R\},s']}\ \mathrm{Rel\ where}\ R = \{a'/a, \cdots\}$$

$$\frac{}{[\mathrm{define}\ A\ P,s] \rightarrow [\mathrm{STOP},s]}\ \mathrm{Def}$$

$$\frac{}{[\mathrm{bind}\ A\ string,s] \rightarrow [\mathrm{STOP},s]}\ \mathrm{Bind}$$

$$\frac{}{[\mathrm{STOP},s] \not\rightarrow}\ \mathrm{Stop}_1$$

$$\frac{[\mathrm{STOP},s] \not\rightarrow\ [\mathrm{STOP},s] \not\rightarrow}{[\mathrm{STOP}||\mathrm{STOP},s] \not\rightarrow}\ \mathrm{Stop}_2$$

$$\frac{[P,s] \overset{\alpha}{\rightarrow} [P',s']}{[\mathrm{STOP}++P,s] \overset{\alpha}{\rightarrow} [P',s']}\ \mathrm{Stop}_3$$

$$\frac{[\mathrm{STOP},s] \not\rightarrow\ [\mathrm{STOP},s] \not\rightarrow}{[\mathrm{STOP}++\mathrm{STOP},s] \not\rightarrow}\ \mathrm{Stop}_4$$

図 8 モデル記述言語の操作的意味
Fig. 8 Operatinal semantics of a model discription language.

クから，最初に現れた a を含む組を削除し，その後のリストを返す．member(a,s) は s の状態にあるスタックに，a を持つ組 $a;v$ が s に存在するかを確かめる関数である．存在すれば真を返し，そうでないときには偽を返す．

(平成 17 年 12 月 14 日受付)
(平成 18 年 3 月 22 日採録)

永藤　直行（正会員）

昭和 43 年生．平成 8 年北陸先端科学技術大学院大学情報工学研究科修士課程修了．平成 8 年三菱マテリアル株式会社勤務．平成 10 年有限会社プレシステム創業，現在在職中．平成 13 年東京工業大学情報理工学研究科計算工学専攻博士後期課程入学，現在在籍中．ソフトウェアセキュリティ，プログラミング言語に関する研究に従事．ソフトウェア科学会会員．