論文 / 著書情報
Article / Book Information

題目(和文)	強制可能なセキュリティポリシーの特徴に関する研究
Title(English)	
著者(和文)	永藤直行
Author(English)	
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第6671号, 授与年月日:2006年9月30日, 学位の種別:課程博士, 審査員:
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第6671号, Conferred date:2006/9/30, Degree Type:Course doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

強制可能なセキュリティポリシーの特徴に関する研究

東京工業大学
情報理工学研究科
計算工学専攻
渡部研究室 01D38086
永藤 直行

平成 18 年度 博士論文

概要

本研究は，ソフトウェアを安全に構成・実行するための機構に関するものであり，その成果は大きく二つにわけられる．一つは実行監視（ソフトウェアの動作を監視しながら実行する技術）に関する理論的成果であり，もう一つはソフトウェアのソースコードを検査するための技術に関する成果である．前者は，実行監視により検査できる性質が従来知られていたものより広くできることを証明し，それによりある種のコバートチャネルを介した情報漏洩が検出できることを示したものである．後者は，広く用いられているデバッカである GDB を用いてソフトウェアのモデル検査に利用したモデルをソースコードの試験にも利用し，検証のコストを相対的に低下できることを示したものである．

モニタとは，実行監視の一種であり，プログラムの実行イベントを監視し，その実行ステップがポリシーに違反するときにはそのプログラムを停止することによってポリシーを強制する仕組みのことである．このモニタを，単にプログラムの実行履歴だけを利用するのではなく，静的解析や観測した実行履歴の部分列から得られる情報も利用してポリシーを強制するように拡張し，この拡張されたモニタを利用してコバートチャネルの出現を実行時に発見できる可能性を理論的に示す．コバートチャネルとはアクセス制御だけでは発見できない隠れた通信路のことであり，この通信路を利用してアクセスポリシーに違反せずに機密情報を流出させることが可能である．コバートチャネルを発見するために，拡張モニタは，観測した実行履歴の部分列による計算資源の状態をエミュレートする付加構造をもち，その状態を付加的な情報として利用する．ここで部分列は，つぎのように計算される．プログラムはそれぞれいくつかのグループにあらかじめ分類されているものとし，それらのグループ間には半順序関係が定義されているものとする．このとき，高レベルのプログラムの実行イベントをすべて除いて得られる実行履歴を部分列とする．この部分列によるエミュレータの状態と実際の計算資源の状態を比較し，それらが異なるときには高レベルのプログラムが低レベルのプログラム

に干渉していると考え，そのとき情報漏洩が起こっていると見なす．この性質は不干渉性として知られる性質であり，コバートチャネルをより良く扱うことができる．一方で，F.B.Schneider は，単に実行履歴だけを利用するモニタのクラスにより強制可能なポリシーの特徴を明らかにしている．このクラスでは不干渉性を強制することはできない．このクラスの監視機構により強制可能なポリシーは単一の実行履歴上の性質を表しており，その特徴から実行列の未来の情報は利用できないとして，強制可能なポリシーは Lamport の安全性の特徴をもつことを示した．しかし，不干渉性は実行履歴の集合上の性質であるので安全性の特徴を充たさず，強制できない．そこで，本稿では拡張モニタによってこの不干渉性が強制できることを示す．

ソースコードの検査では，従来から知られているモデル検証の手法を利用する．一般的に，ソフトウェアのライフサイクルには二つの検証工程がある．一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，もう一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，システムの状態空間を形式的に検証する．既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．

ABSTRACT

This study discusses two topics in order to build high assurance secure systems. We begin by discussing the characterization of security policies making use of a monitor to enforce security policies. A monitor is an enforcement mechanism that works by observing the runtime behavior of a program and terminating its execution if it violates a security policy. We introduce a monitor which is extended from Schneider's execution monitor known as security automata. The extended monitor can make use of more information such as a set of subsequences to be observed and can enforce O'Halloran's noninference as an information flow property. Thus it can detect covert channels via shared storage at run time.

We then use model checking techniques in order to examine if a system satisfies properties which are unenforceable by the monitor and discuss a model checking tool which directly examines source code. Most systems are checked at one or both of two parts of the software life cycle; one is verification that the models used actually describe the functional specification, and the other is verifying that the source code implements the model. There are several model checking tools that handle either the verification of models or source code, but not both. We describe a model checking tool can be used in both cases. The tool executes the implementation under verification using GDB, and examines a composition with the model in which portions corresponding to the source code has been eliminated. It enables the use of the same model which was used in the verification process and thus reduces the cost of implementation. We finish off by providing a simple ECHO service example.

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　セキュリティポリシーとその強制機構

コンピュータセキュリティとは，不慮のあるいは悪意の暴露や改竄から計算資源を保護することである．このようなコンピュータセキュリティの特性は，一般に完全性，機密性，可用性とよばれる．これらは，以下の三つのことについて議論している．

- 完全性 (integrity)：データやプログラムを明確な権限がある方法によってのみ変更や消去ができることを保証する．
- 機密性 (confidentiality)：機密情報が権限のない受信者によって暴露されないことを保証する．
- 可用性 (availability)：計算資源を，権限のあるユーザが必要としたときにはいつでも利用できることを保証する．

これら三つの特性がどの程度必要であるかは，システムが利用される状況によって異なる．たとえば，防衛産業ではおもに機密性が重要視される．対象的に，電話産業では可用性がもっとも重要視されるであろう．あるシステムが必要とする要求は，そのシステムや業務のセキュリティポリシーで述べられる．

システムのセキュリティを保証するために手続き的な方法や付加的な機構が導入されてきた．いくつかのセキュリティについての用件は，従うべき手続きやそれを強制する機構によって実現される．たとえば，適切なパスワードを選択するための用件を考える．このとき，パスワードは大文字，小文字，数字さらに特殊記号を用いて 8 文字以上でなければならない．これは，パスワード管理における一種のポリシーであり，パスワードを作成，変更するシステムの機構によって，このポリシーを強制することができる．システムのセキュリティをさらに高めるために，認証機構によりこのパスワードを用いてユーザの同定を行ったのち，正当

なユーザが正当な計算資源にのみアクセスできることを保証する．これをアクセス制御という．このとき，セキュリティポリシー（アクセス制御ポリシーともいう）はアクセス行列 (access matrix) として定義され，参照モニタという機構を利用してこのポリシーを強制する．アクセス制御ポリシーを説明するために，主体，客体の概念を導入する必要がある．主体とは，能動的な実体であり（たとえば，ユーザプロセス），客体とは，受動的な実体である（たとえば，メモリ，ファイル，プログラム）．プログラムのような実体は，客体でもあり主体でもある．アクセス制御ポリシーは，各主体がそれぞれの客体にどのようなアクセス権限をもっているかを決定している．アクセス権限とは，たとえば，読込み，書込み，実行権といったものである．権限を決定する方法により，任意アクセス制御と強制アクセス制御に分類できる．正確には，以下のように述べられる．

- 任意アクセス制御 (Discretionary Access Cotrol:DAC)：客体の所有者がその客体に他の主体がどのタイプの権限を許すか指定できる．
- 強制アクセス制御 (Mandatory Access Cotrol:MAC)：客体と主体のセキュリティ属性を基準に，客体へ主体がどのようなタイプの権限を許すかシステムが決定する．

アクセス制御には，さらにつぎのような機構がある．移動コードには極端に制限された権限のみを許すアクセス制御で，これをサンドボックスという．このほかに，PCC(Proof Carrying Code) [24]，SASI(Security Automata Software Isolation) [8]，MCC(Model Carrying Code) [30] という強制機構も報告されている．アクセス制御リストが単一の実行ステップについてのポリシーであるのにたいして，SASI は実行ステップの時系列も考慮したポリシーを許し，それを強制する機構である．セキュリティポリシーを可能な実行ステップの時系列の集合をオートーマトンで表し各実行ステップを観測してそのオートマトンにより受理されるか否かを判定する．この概念を 図 1.1 に示す．この図ではセキュリティポリシーはプログラムを変換してプログラムコード内に埋め込まれている．

近年では，多くのシステムが不特定多数の利用者が利用するネットワークに接続されており，侵入検知機構や防火壁も利用されている．これも，IP アドレスやポート番号による一種のアクセス制御である．これらのようにシステムの各実行ステップを監視し，もしポリシーに違反するならば強制的に停止させることでポリシーを強制する機構のクラスを実行監視とよぶことにする．

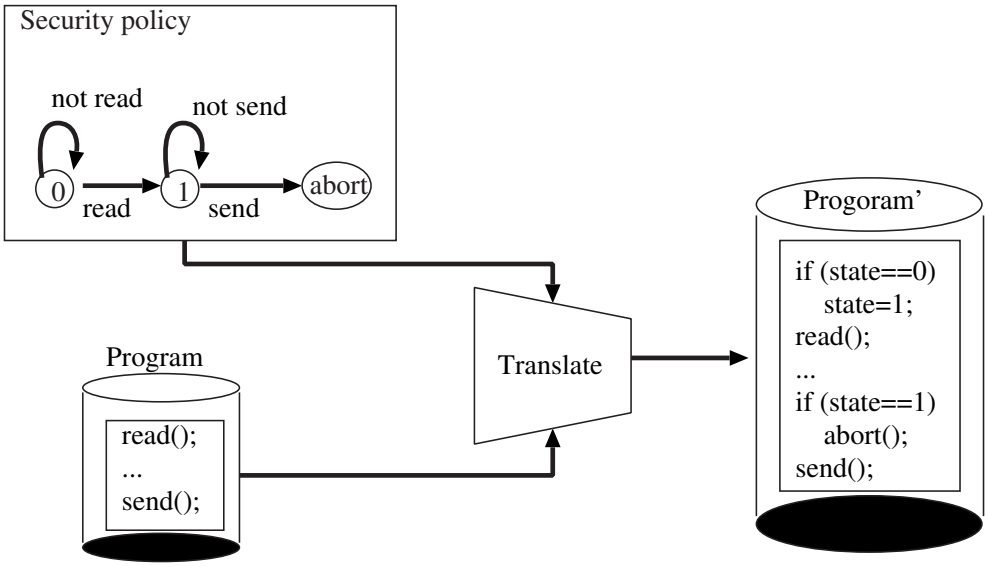

図 1.1: SASI の簡単な例

この節では，さまざまなセキュリティポリシーとその強制機構について述べてきた．これ以降は，実行監視によって強制されるポリシーの特徴について議論する．

1.2 システムセキュリティ評価基準

システムのセキュリティを評価することは非常にむずかしい問題である．セキュリティシステムのテストの評価は，タイガーチームを利用するという試みがある．このチームは，設計や実装の不備を利用することで，不正アクセスを試みようとする．また，NCSC(National Computer Security Center) により，機密性に関する評価基準のひとつである TCSEC [6] が定められた．この基準では，大きく四つのディビジョンにわかれており，それぞれ A,B,C,D と名付けられ，それぞれのディビジョンはいくつかのクラスにさらに分類される．A がもっとも厳しい基準であり， D がもっともゆるい基準である． D は A から C までの基準を充たせなかったシステムがここに分類される．C は任意アクセス制御を行うように要求している．B は三つのクラスに分類されており，B1 クラスは，任意アクセス制御に加え，強制アクセス制御を行うように要求してる．さらに B2， B3 クラスでは，コバートチャネル解析 (CCA) [4, 7] を行なうことを要求し

ている．A は，要求してる要件は B3 と同じであるが，各要件を充たしていることを形式的に示すことを要求している．

この基準から，コバートチャネルを発見することは， B2 以上の安全なシステムを開発するときには重要である．

1.3 コバートチャネルとセキュリティモデル

1.1 節でアクセス制御について述べた．機密性を保証するためにはアクセス制御だけでは不十分であることがすでに知られている．アクセス制御ポリシーを正確に強制できたとしても，トロイの木馬のようなプロセスはコバートチャネルを作成し，そのポリシーに反して情報を漏洩することができる [7]．コバートチャネルとは，直感的には，システムで想定していない通信路のことである．以下では，このことについて議論する．

まず，アクセス制御モデルについて述べる．DAC, MAC それぞれにアクセス制御モデルが存在する．ここでは，BLP モデル [3] を利用することにする．アクセス権限は r:read と w:write のふたつとし，主体，客体の集合をそれぞれ S, O であるとする．BLP でもポリシーはアクセス制御行列 M によって表される．まず，主体と客体にセキュリティレベルという適当な属性をあたえる．この属性の集合を L とし，その上で半順序関係 $>$ が定義されているものとする．さらに，ある主体が客体へ現在許されている操作を

(s,o,a)

の集合 C として表す．ここで，$s \in \mathsf{S}, o \in \mathsf{O}, a \in \{r, w\}$ とする．主体，客体とセキュリティレベルの間には，つぎの関数 $F : \mathsf{S} \cup \mathsf{O} \to \mathsf{L}$ が定義されていて，現在の状態での主体または客体のセキュリティレベルをかえす．このモデルでは，システムの現在の状態は (C, F, M) によって表される．Bell と LaPadula [3] はシステムのセキュリティ関する用件をつぎの二つの定義により結論づけた．ひとつは simple security property とよび，もうひとつは， *-property とよぶ．

定義 1.1 (ss-property) どんな $s \in \mathsf{S}, o \in \mathsf{O}$ についても，$(s, o, r) \in \mathsf{C}$ で $r \in M(s, o)$ ならば $F(s) > F(o)$ または $F(s) = F(o)$ である．このとき，状態 (C, F, M) は読出し安全であるという．

直感的には，セキュリティレベルの低い主体が，レベルの高い客体の情報を読出すことを禁止している．

定義 1.2 (*-property) どんな $s \in \mathsf{S}, o \in \mathsf{O}$ についても，同時に $(s, o, r) \in \mathsf{C}$ かつ $(s, o', w) \in \mathsf{C}$ であるとき $F(o') > F(o)$ または $F(s) = F(o)$ である．このとき，状態 (C, F, M) は書込み安全であるという．

直感的には， *-prprety はシステムの遷移そのものが安全でなければいけないことを表している．トロイの木馬のようなプログラムがセキュリティレベルの高い客体の情報をレベルの低い客体に書込むことを禁止している．ここで，その状態が ss-property と *-property を充たしているとき，その状態は状態安全であるということにする，また，初期状態が状態安全で，そこから有限回の遷移によって到達可能な状態のすべてが状態安全であるときシステムは安全であるとする．Bell と LaPadula [3] は，安全なシステムに関するつぎの定理 (basic security theorem) を示した．

定理 1.1 (BST) システムがつぎの二つの条件を充たすとき安全である

1. 初期状態が状態安全で，かつ，
2. ある状態 (C, F, M) つぎの状態 (C', F', M') として (s, o, a) が C に追加されるときにはいつでも
   - *(a)* $(s, o, r) \in C' - C$ ならば $F'(s) \geq F'(o)$.
   - *(b)* $(s, o, r) \in C$ かつ $F'(s) \not\geq F'(o)$ ならば $(s, o, r) \notin C'$.
   - *(c)* $(s, o, w) \in C'$ かつ $(s, o, w) \notin C'$ ならば $F'(s) \geq F'(o)$.
   - *(d)* $(s, o, w) \in C$ かつ $F'(o) \not\geq F'(s)$ ならば $(s, o, w) \notin C'$.
   - *(e)* $(s, o, a) \in C' - C$ ならば $a \in M'(s, o)$.
   - *(f)* $(s, o, a) \in C$ かつ $a \notin M(s, o)$ ならば $a \notin M'(s, o)$.

ここまで，システムが安全であるということについて述べた．つぎにこのモデルの問題点について述べる．

BLP モデルは，直感的であり容易に実現できる．しかし，すべての計算資源を主体と客体としてアクセス権限を定義することはそれほど簡単ではない．たとえば， CPU ，通信回線，システムコールなどの基本的なプログラムアクセス権限を正確にすべてあたえることはむずかしい．このことが原因でトロイの木馬のようなプログラムはコバートチャネルを

構成することができてしまう．具体的には以下で述べるようにコバートチャネルを構成できる．たとえば，Holly と Luch は，二つのユーザプロセスとし，セキュリティレベルは H, L で $H > L$ とする．アクセス制御ポリシーは，彼らはそれぞれのレベルと同じファイルやディレクトリへは読込みと書込みが許されている．自身がオーナーであるディレクトリを子に持つディレクトリをお互いのディレクトリの子として作成できる，とする．また，UNIX のそれと同じ意味を持つ remove と create のような TCB(trusted computing base) を利用できる．このとき，二人のユーザは，図 1.2 の Upgraded Directory のような共有資源を利用して Holly が送信者で Lucy が受信者となるコバートチャネルを作成できる [7]．Lucy は H のレベルに upgraded directory を作成し， Holly はそのディレクトリの下にファイルを作成したり削除したりする．Lucy はそのディレクトリにたいして remove 操作を試み，その結果を観測し， 0 または 1 という 1 ビットの情報を獲得することができる．図 1.2 は remove が成功する場合を表しており，図 1.3 は remove が失敗する場合を表している．この簡単な例では， remove の意味を変更すれば解決するようにみえるが，このような基本的な操作の意味を変更することは簡単ではない．あるアプリケーションは remove のこの意味を利用しているかも知れないからである．よって，コバートチャネルを作成する方法が一度見付けられてしまうとそれを解決することも発見することも容易ではない．また，コバートチャネルは様々なレベルで計算資源を共有することによって生じる．さまざまな共有資源すべてに主体，客体としてアクセス権限を与えることは非常にむずかしい．このことが原因でコバートチャネルは生じる．もちろん、コバートチャネルを構成する方法は多岐にわたっており、前述の方法以外にも多くの方法がある．利用する共有資源に着目して分類すると共有記憶チャネルとタイミングチャネルに分類できる．不干渉性は，システムの入出力関係の制限を定めた機密性モデルでありインタフェイスモデルとよばれている．このモデルは，コバートチャネルの問題をよりよく扱うことができる．つぎに，このインタフェイスモデルについて述べる．あとで述べる不干渉性を用いたコバートチャネルを発見する方法は共有記憶チャネルを扱えることが知られている [4, 7].

インタフェイスモデルは，セキュリティを強制する方法を定めるというよりは，安全でないシステムを回避するための入出力関係を定めたものである．この関係を不干渉性として定め，システムが安全であるための十分条件を与える．ここでは，コバートチャネルを発見するために利

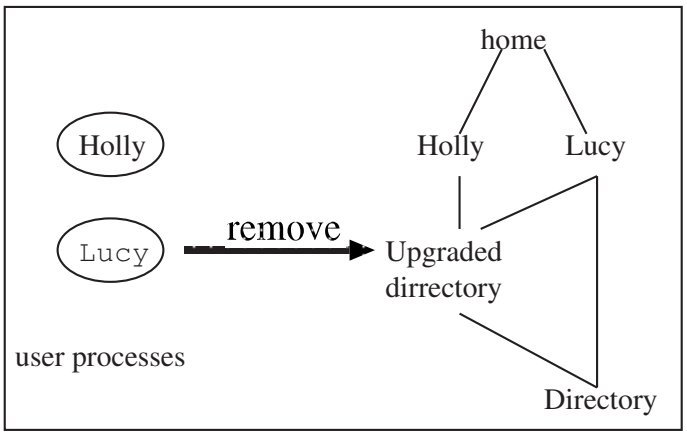

図 1.2: remove が成功するときの例

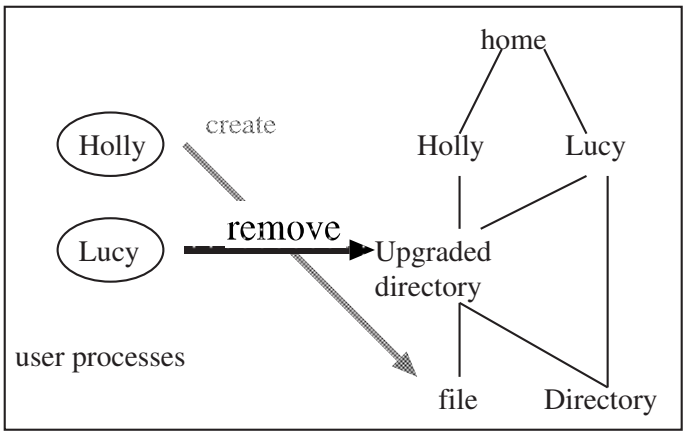

図 1.3: remove が失敗するときの例

用する [7, 19]．不干渉性は，高レベルユーザの入出力が低レベルユーザの振舞に干渉しないという概念を形式化したものである．この形式化は，Goguen と Meseguer [13] によって最初になされた．この方法は，システムの状態の観測が各ユーザによって可能であることを必要とする．どんな入出力の列を実行したときにも，あるユーザによって観測されるシステムの状態がある入出力をその列から削除して得られる列を実行したときのシステムの状態と同じであるとき，そのシステムは安全であるという．ユーザ u へのシステムからの出力が $out(u, h, c)$ によってあたえられる決定性システムを考えている．ここで，h はこのシステムへの入力履歴で c は h につづけて u が実行するコマンドを表している．システムのセ

キュリティは入力ヒストリのパージによって定義される.

定義 1.3 (不干渉性) $purge$ を以下の条件を充たす ユーザ × ヒストリ → ヒストリ の関数とする.

- $purge(u, <>) =<>$，ただし $<>$ は空のヒストリを表す.
- $purge(u, h \cdot c(w)) = purge(u, h) \cdot c(w)$，ただし $c(w)$ はユーザ w により実行されるコマンドを表し， $F(u) \geq F(w)$ であるとする.
- $purge(u, h \cdot c(w)) = purge(u, h)$，ただし $c(w)$ はユーザ w により実行されるコマンドを表し， $F(u) \not\geq F(w)$ であるとする.

すべてのユーザ u すべてのヒストリ h そしてすべてのコマンド c について

$$out(u, h, c(u)) = out(u, purge(u, h) \cdot c(u))$$

のとき，システムは不干渉であるという.

つぎに，システムがこの条件を充たすことを確認する手段について述べる.

Goguen と Meseguer [14] はシステムが不干渉であるための十分条件を示した．これは Unwinding Theorem として知られている．不干渉性は，システムのヒストリに関する条件を述べている．一方で，Unwinding Theorem はそれぞれの実行ステップに関する条件を述べている. ここで、再びシステムのモデルとセキュリティポリシーについて以下のようにモデル化する．ここでは， Rushby [27] の表記を用いる．システムを状態遷移マシンと考え，その状態遷移マシンはつぎのものからなるとする.

- 集合 S は状態の集合
- 集合 C はコマンドの集合
- 集合 Out はコマンドの出力の集合
- 関数 $out : S \times C \rightarrow Out$ は与えられたユーザが状態 $s \in S$ で観測するものを表している．これを output function という.
- 関数 $step : S \times C \rightarrow S$ は，状態 $s \in S$ をコマンド $c \in C$ によって更新する方法を表している．これを state transition function という.

- 関数 $s_0 \in S$ は初期状態である．

さらに，関数 $run : S \times A^*$ を導入する．これは $step$ のコマンド列（ここではヒストリとよぶ）への自然な拡張である．

$$\begin{aligned} run(s, \epsilon) &= s \\ run(s, a \cdot \alpha) &= run(step(s, a), \alpha) \end{aligned}$$

関数 run を上述の等式によって定義する．ここで，ϵ は空のヒストリを表し，$\cdot$ はコマンドと列の連結を表し，$a \in C$ とする．状態を表すために $s, t, \cdots$ を用い，コマンドを表すために $a, b, \cdots$ を用い，コマンド列を表すために $\alpha, \beta, \cdots$ を用いることにする．システムだけでなくセキュリティポリシーもモデル化する．システムセキュリティを議論するために，セキュリティドメインの集合とそれらドメイン間の情報の許される流れを制限するポリシーを仮定する必要がある．また，

- 集合 D はドメインの集合，
- 関数 $dom : C \to D$

とする．セキュリティポリシーは，D 上の反射的な二項関係 $\rightsquigarrow$ によって指定される．また，$\not\rightsquigarrow = D \times D - \rightsquigarrow$ とする，ここで $-$ は差集合を表す．

定義 1.4 (view-patition) あるシステム M が *view-patitioned* であるとは各 $u \in D$ について S 上の関係 $\overset{u}{=}$ が存在することである．

$$s \overset{u}{=} t \Rightarrow out(s, a) = out(t, a)$$

であるとき，これらの関係を *output-consistent* であるという．

直感的には，あるドメインから同じにみえる状態は必ずどんなコマンドの出力も同じになるということである．不干渉性において，システムのセキュリティの定義は，あるドメインによって観測される出力が他のドメインのアクションによって影響されないということである．つぎの補題はこのことを表している．

定義 1.5 (Multi Level Secure) M は *view-patitioned* ，ポリシー $\rightsquigarrow$ とする．

$$out(\alpha, a) = out(purge(dom(a), \alpha), a)$$

であるならば M は $\rightsquigarrow$ について安全であるという．

補題 1.1 ポリシー $\rightsquigarrow$，システム M が *view-patitioned* であり，システム M が

$$do(\alpha) \stackrel{u}{=} do(purge(\alpha, u))$$

となる *output-consistent* システムであるとする．そのとき， M はポリシー $\rightsquigarrow$ について安全 *(secure)* である．ここで，関数 $do : C^* \rightarrow S$ を $do(\alpha) = run(s_0, \alpha)$ とする．

証明 1.1 この証明は， *[27]* の *lemma 1* の証明と同じである．文中の u を $u = dom(a)$ として

$$do(\alpha) \stackrel{dom(a)}{=} do(purge(dom(a), \alpha))$$

とする． *output consistency* であることより

$$out(do(\alpha), a) = out(do(purge(dom(a), \alpha)), a)$$

MLS の定義より M は安全である．

定義 1.6 M を *view-patitioned* システム， $\rightsquigarrow$ をポリシーとする．

$$dom(a) \not\rightsquigarrow u \Rightarrow s \stackrel{u}{=} step(s, a)$$

ならば M は *locally respect* $\rightsquigarrow$ であるという．そして，

$$s \stackrel{u}{=} t \Rightarrow step(s, a) \stackrel{u}{=} step(t, a)$$

ならば M は *step consistent* であるという．

M が locally-respect $\rightsquigarrow$ であるとは，特定のコマンドを消去したヒストリについて述べており，M が step consistent であるとは，ドメイン u が観測する出力に干渉するコマンドの列について述べている．

定理 1.2 (Unwinding Theorem) $\rightsquigarrow$ をポリシー， M を *view-patitioned* システムとする．M が *step consistent* かつ *locally respect* $\rightsquigarrow$ であるならば M は $\rightsquigarrow$ について安全 *(secure)* である．

証明 1.2 $s \stackrel{u}{=} t \Rightarrow run(s, \alpha) \stackrel{u}{=} run(t, purge(u, \alpha))$
であることを α の長さに関する帰納法を用いて示す．

Basis $\alpha = \epsilon$ のとき $purge$ の定義より $s \stackrel{u}{=} t \Rightarrow run(s, \alpha) \stackrel{u}{=} run(t, \alpha)$ となり，明らか.

Inductive Step 長さ $\alpha = n$ について仮定し，$\alpha = a \cdot \alpha$ を考える．run の定義より

$$run(s, a \cdot \alpha) = run(step(s, a), \alpha)$$

となる．$run(s, purge(u, a \cdot \alpha))$ について場合分けする.

Case 1: $dom(a) \rightsquigarrow u$ のとき，*purge* の定義より

$$\begin{aligned} run(t, purge(u, a \cdot \alpha)) &= run(t, a \cdot purge(u, \alpha)) \\ &= run(step(t, a), purge(u, \alpha)) \end{aligned}$$

M は *step consistent* であり， $s \stackrel{u}{=} t$ であるので

$$step(s, a) \stackrel{u}{=} step(t, a)$$

であり，よって，帰納法の仮定より

$$run(step(s, a), \alpha) \stackrel{u}{=} run(step(t, a), purge(u, \alpha))$$

Case 2: $dom(a) \not\rightsquigarrow u$ のとき，*purge* の定義より

$$run(t, purge(u, a \cdot \alpha)) = run(t, purge(u, \alpha))$$

M は *locally respect* $\rightsquigarrow$ であり， $dom(a) \not\rightsquigarrow u$ であるので

$$s \stackrel{u}{=} step(s, a)$$

$s \stackrel{u}{=} t$ で $stackrelu=$ が同値関係であるので

$$step(s, a) \stackrel{u}{=} t$$

このことと帰納法の仮定より

$$run(step(s, a), \alpha) \stackrel{u}{=} run(t, purge(u, \alpha))$$

を得る.

ここで，$s = t = s_0$ として

$$do(\alpha) \stackrel{u}{=} do(purge(u, \alpha))$$

となり，補題 *1.1* よりしたがって，定理 *1.2* は正しい． ■

この方法は，不干渉性ポリシーを強制したいシステムを検証する基本原理を与える．第 2 章で実行時にポリシーを強制できることを示すためにこの方法を利用する．ここまで，システムは決定的であると仮定していた．この節の残りの部分では，システムが非決定的である場合について述べる．

決定性システムを考えるときには前述の不干渉性はその条件が強すぎるという意味で，実用上は完璧である．一般に，プログラムは決定的であると考えられるが，システム仕様が決定的であることを望むのは実際的でないので，不干渉性を非決定性システムに応用することを考える．はじめに，非決定性システムをモデル化することを試みる．out は関数ではなく関係とする．すなわち，同じ入力にたいして異なる出力を許すことにする．しかしながら，入出力の関係を保存するためにはヒストリは入力だけでなく出力もヒストリ自身に含めなければならない．また，出力が同じであるという概念のかわりにヒストリのパージもまた受理トレースであるという概念を用いる．しかし，この方法では，不受理がセキュリティ違反によるものなのか，セキュリティ以外のシステム要件によるものなのか区別がつかない．たとえば，前述の $purge$ は入力コマンドだけを取り除いているので出力コマンドは対応づけられる入力コマンドが存在しなければいけないシステムでは，システム要件による不受理であるが，セキュリティ違反による不受理とこのことは区別がつかない．そこで，$u \not\rightsquigarrow v$ であるならば，$dom(a) = u$ となるすべての入出力コマンドを取り除く $purge$ を再定義し（Noninference [25]），この $purge$ により生成されたヒストリもまた受理トレースであるという条件を考える．しかしながら，この条件は強すぎる．二つのセキュリティドメイン H, L，ポリシー $L \rightsquigarrow H$ とし，$dom(l) = L$ が $dom(h) = H$ を生成するようなシステムとすると，再定義した $purge$ によりすべての H になる入出力がすべて取り除かれるために，このシステムは安全でないことになる．さらに，深刻なのは安全でないシステムを許してしまうことである．$h_{in}, h_{out}, l_{in}, l_{out}$ を入出力イベントとし，可能なトレースが

$$\{<>, h_{in}(0), h_{in}(1), l_{in}(0), l_{in}(1), h_{in}(0)\ l_{out}(0), h_{in}(1)\ l_{out}(1)\}$$

であるシステムを考える．ここで，$<>$ は空列を表す．このシステムでは，すべてのトレースから h_{in}, h_{out} を取り除いたトレースもまた受理ト

レースとなってしまう．このようなシステムでは l_{out} のまえに適当な h_{in} を挿入するだけで $H \rightsquigarrow L$ となるシナリオができてしまう．この問題は，同じ入力で異なる出力を許すようなシステムでは，パージしたトレースもまた受理トレースであるという条件だけでは弱すぎることである．そこで，受理トレースに任意の高レベルイベント h_{in} あるいは h_{out} を挿入したトレースも受理トレースであるとする (Separability)．しかしながら，この条件では強すぎ，各ドメイン間で情報を流すことができなくなってしまう．高レベルイベント自体は受理可能であり，しかもこの高レベルイベントは低レベルイベントとともに高レベルの出力を変えうることを考慮する必要がある．このことから、ふたつの受理トレース T,S にたいし，T から高レベルイベントを除いた低レベルイベント列と，S から低レベルイベントを除いたの高レベルイベント列から，順序はそのままで構成されたイベント列に，さらにそして T の低レベルイベントでも S の高レベルイベントでもない任意のイベントを挿入して構成されるイベント列もまた受理トレースである，という条件が導かれる．この性質は，非演繹性 (Nondeducibility) [31] として知られている．非演繹性は，決定性を仮定していないので不干渉性よりは一般的であるが，ユーザが二人だけのとき，不干渉性と等価である．しかし，二人以上のとき不干渉性より真に弱い．

非演繹性には，ふたつの問題がある．ひとつはそれが弱すぎるであり，もうひとつはシステムの合成に関して閉じていないことである．非演繹性の定義を振り返ると，これらの問題は，ふたつの受理トレースから他のトレースを生成するときにあまりにも多くの自由を許していることに起因する．T と S の間でイベントの順序が保存されるだけでなく，どのように組み合わせるかということに関して制限を設けるべきである．実際に，適当な部分に高レベルイベントを挿入したものも受理トレースであるという条件ではなく，つぎの条件が導かれる．受理トレース T から高レベル入力を挿入または削除を行なったトレース T' が与えられたとき，T' の T から変更を加えられた部分よりあとの箇所に高レベル出力を挿入または削除を行なって得られるトレース T" もまた受理トレースである，という条件を導く．これは，一般化不干渉性 (Generalized Noninterference) [18] として知られている．しかしながら，一般化不干渉性では，まだ合成に関して閉じていない．そこで，合成可能なセキュリティ特性を導くためにさらに T" の作り方に制限を加える．受理トレース T が与えられ変形 T' が T から高レベル出力を追加または削除を行なったとき，T' を作るた

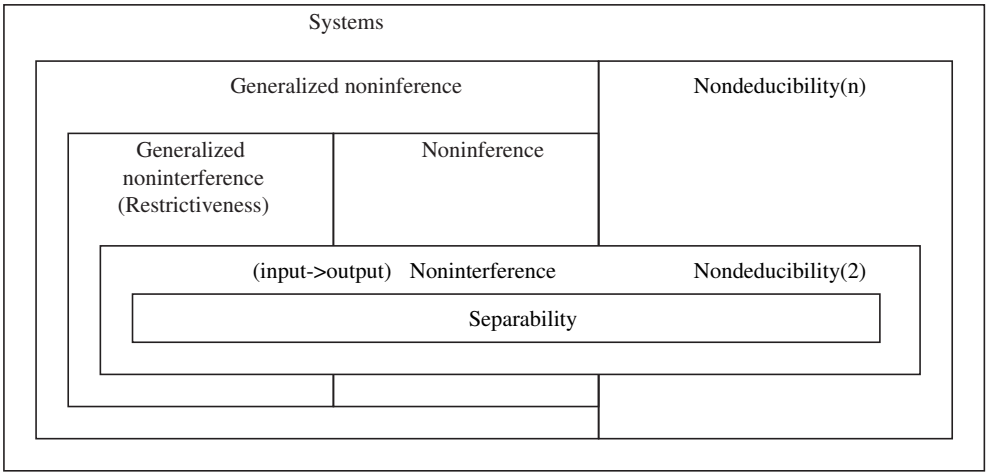

図 1.4: 情報流の性質の分類

めに T から変更が加えられた部分よりあとの箇所に高レベル出力を挿入または削除したトレースと，T にたいし変更を加えた部分の直後の低レベル入力の任意のトレースからなるトレース T” もまた受理トレースである，という条件を考える．これは制限性 (Restrictiveness) [18] として知られている．たとえば，受理トレース wxyx について考える．ただし，z は高レベル出力，x,y は低レベル入力である．ここで，高レベル入力 h を w の直後に挿入し，whxyz を作る場合を考える．一般化干渉性では whxyz から z を削除することもしくは h よりあとの部分ならどこにでも高レベル出力を加えることができる．制限性は，変更部分の直後に低レベル入力が続かねばならないので，z を削除することと y よりあとの部分に高レベル出力を挿入することを制限している．これらの性質の間には図 1.4 に示される包含関係がある [20, 33].

1.4 本研究の位置づけ

本稿は，実行監視によって強制可能なセキュリティプロパティの特徴について議論している．我々の研究が貢献したであろうことを以下に概説する．

- エミュレータをもつ実行監視により情報流に関するセキュリティポリシーが強制可能であることを示した．これにより，実行時にコバー

トチャネルを発見することが可能になり，より安全なシステムを構成することが可能になる．このことは，第 2 章で述べられる．

- システムを安全に構成するためにソースコードレベル検証器も開発した．システムモデルを試験環境とし，デバッカ上で直接，モデルと同期をとりながら実行コードを実行することでソースコードを直接検証する．このツールについては，第 3 章で述べられる．
- プロセス代数にもとづいたポリシー記述言語を提案し，その構文と意味を定義している．この言語は，ソースレベル検証器ではモデル記述言語としても用いられている．第 4 章で述べられる．

第 2 章と第 3 章の関係についてここで述べる．つぎの観点から本研究では，ソースレベルモデル検証器も開発している．

- 静的なモデル検証から実行監視までをひとつのツールにより実現する．
- 強制機構は，静的なモデル検証することができない性質をセキュリティポリシーとして与える必要がある．

二つめの項目について，もう少し付け加える．モデル検証において，状態数爆発という問題を解決するために，たとえば， partial order reduction という手法を利用したとする．このとき線形時間論理は next 演算子を除いたものになる．もし強制可能なセキュリティポリシーにこの演算により表されるシステムに要求される性質が含まれていれば，静的なモデル検証を補うかたちで実行監視を用いることが可能である．この観点からセキュリティポリシーの特徴を整理することは有意義であるとおもわれる．第 3 章の検証器は，そのための予備実験としても位置づけられる．

1.5 本稿の構成

以下では，本稿の構成について述べる．第 2 章では，付加的な構造を実行監視機構に追加することで情報流に関するセキュリティポリシーが強制可能であることを述べる．この章では，さらに我々が開発した強制機構を概観する．ソフトウェアを安全に構成するためには動的に監視するだけでなく，静的に検査する必要がある．そこで，第 3 章では，静的にソフトウェアを解析する検証器について述べる．これは，我々が開発し

たソースレベル検証器である．第 4 章では，セキュリティポリシー記述とモデル記述において共通に利用する記述言語について述べる．第 5 章では，全体のまとめと未解決の問題について述べる．

第2章　拡張モニタにより強制可能なポリシー

2.1　はじめに

実行監視（Execution Monitoring）とは，プログラムの単一の実行ステップを観測し，もしその実行ステップが，管理者によって与えられたセキュリティポリシーに違反しているときにはそのプログラムを停止または変更することによりポリシを強制する機構のクラスである．以降，この機構を**モニタ**と呼び，観測対象であるシステムやプログラムを単にターゲットと呼ぶことにする．もちろん，このクラスの機構はすべてのセキュリティポリシーを強制することができるわけではない．どのようなセキュリティポリシーをモニタは強制することができるだろうか．F.B.Schneider [29] はこの問題について最初に議論し，その特徴を明らかにした．彼は，モニタの特徴からそれは単にターゲットの観測可能な有限の振舞だけが利用可能であるとし，そのようなポリシーは Lamport の安全性の特徴を充たすことを示した．また，モニタとしてそのような安全性を充たすポリシーを受理する Büchi オートマトン [1] を考え，それをセキュリティオートマトンと呼んでいる．このオートマトンでは，単にプログラムの実行列だけを利用している．我々はこのオートマトンを拡張し，より多くの情報を利用するモニタについて考える．拡張されたオートマトンでは，各状態は実行列の集合によりラベル付けされている．我々の目標はこのモニタにより強制されえるポリシーの特徴をあきらかにすることであるが，本稿では，そのはじめの試みとして情報流ポリシーがある情報流に関する性質（Information flow propeties）のもとで強制可能であることを示す．**情報流に関する性質**とは，実行列の集合についての性質であり，一般には，コバートチャネルの解析の際に利用される性質である [4, 7]．ここで，**コバートチャネル**とは，セキュリティ管理者が考慮していない方法によって，あるユーザから他のユーザへ機密情報を漏洩させるために確立される通信路のことである．たとえば，つぎのように構成することができる．

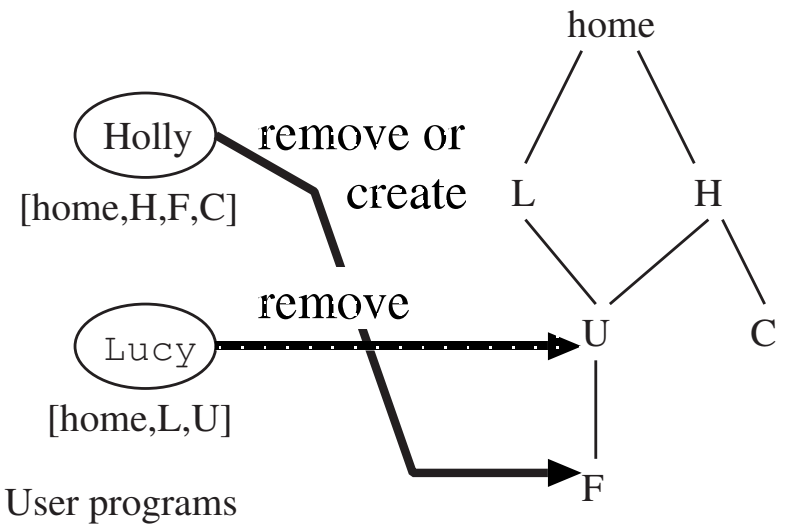

図 2.1: コバートチャネルのシナリオ

いま，図 2.1 の場合を考えることにする．Holly と Lucy を二つのユーザプログラムであるとし，あるファイルシステムにおいて，Holly だけが機密情報 C へのアクセス権限をもち，それ以外へのアクセス権限は [] 内に示されたファイルやディレクトリへのアクセス権限をもつとする．ディレクトリ U はファイルシステムからは異なるディレクトリとして管理される．U は Upgraded Directory [7] や i-node テーブルのようにより下位のレベルで共有されているものとする．また，彼らは UNIX コマンドと同様な remove と create という信頼できるコマンドを実行することができるとする．二つのユーザプロセスは，U を用いて Holly が送信者で Lucy が受信者であるコバートチャネルを生じさせることができる．Holly はディレクトリ U の下でファイル F を作成または削除する．Lucy がそのディレクトリにたいして remove 操作をおこなったとき，その操作は F が存在するかどうかに依存して失敗または成功という結果を得る．この結果をそれぞれ 0 または 1 という 1 ビットの情報とすれば，Holly は機密情報を適当に符号化して Lucy に送信することができる．情報流に関する性質は，コバートチャネルを発見する良い性質であることが知られている．コバートチャネル解析ではシステムの実行列がこの性質を充たしているか否かを検査する．しかし，この性質は詳細化について閉じていない [18]．よって，開発の各段階でコバートチャネル解析が必要である．そこで，我々は，実行監視を用いて実行時にコバートチャネルを発見する方法につい

て議論する．このとき，モニタは従来の構造に加え，付加的な構造を持つ．付加的な構造は観測した実行列の部分実行列を実行したときのシステムの振舞を模倣するエミュレータとなる．このモニタは，アクセス制御といっしょに行なわれることで一部の情報流に関する性質に関してシステムを安全に運用することができる．

本稿の構成を以下に述べる．2.2 節では，強制可能なポリシーの特徴に関する関連研究について述べる．2.3 節では，Schenider の定義にしたがって強制可能なセキュリティポリシーのクラスについて述べ，2.4 節では，我々が行なうモニタの拡張を述べる．2.5 節では，いくつかの実行列上の演算などの定義を与えたあと，一般的な情報流に関する性質を説明する．2.6 節では，本稿で利用する情報流に関する性質を定義し，それが強制可能であることを示す．2.9 節では，本稿のまとめとして理論的な問題点を述べる．

2.2 関連研究

実行監視では，すべてのセキュリティポリシーを強制することができるわけではない．では，どのようなセキュリティポリシーを強制することができるだろうか．F.B.Schneider [29] はこの問題について最初に議論し，その特徴を明らかにした．彼は，モニタの特徴からそれは単にターゲットの観測可能な有限の振舞だけが利用可能であるとし，そのときポリシーは以下の特徴をもつと考えいる．

- 一度ポリシーに違反した実行履歴は，その後もポリシーを充たすことはない．
- ポリシーに違反するときには，観測した有限の実行履歴だけから判断できなければならない．

そして，そのようなポリシーは Lamport の安全性の特徴を充たすことを示した．また，モニタとして，そのような安全性を認識する Büchi オートマトン [1] を考え，それをセキュリティオートマトンと呼んでいる．つまり，プログラムの観測履歴だけを用いるモニタではポリシーに含まれる安全性の部分だけが正確に強制可能であるということである．

しかし，一般にはより多くの情報を利用するモニタを考えることができる．L.Bauer [2] らは，モニタを単なる実行列の認識器としてではなく，

その実行列の変換器と考え Edit Automata を提案した．Edit Automata は，ポリシーに違反するとき，すなわち，なにか悪いことがおこったときただちにプログラムを停止させるのではなく，プログラムの振舞を変更する．たとえば，有限の実行列でなにか悪いことがおこったとき，適当な実行ステップ待つようにするモニタである．そのようなモニタでは上限つき可用性を強制可能であることを彼らは示している．プログラムが計算資源を獲得し，ある時間内にその計算資源を開放するといった性質が上限つき可用性に含まれる．

彼らは，実行列を変更するということを

- Insertion function
- Suppression function
- Editing function

の三つと考えている．Insertion function は，プログラムの状態と実行ステップから適当な部分列を生成し挿入する関数であり，これはポリシー内で定義される．上述の計算資源を開放するポリシーにおいては，計算資源の開放がある時間内に行われるものとする実行列を計算資源の獲得を観測したときに挿入する．時間内に開放されないときには，資源を開放しプログラムを強制的に停止する．Suppression function は，単に観測した実行ステップをなかったものとして削除する．Editing function は， insertion function と suppression function の機能をもち挿入/削除を行なう．Editing function をもつように拡張されたセキュリティオートマトンが Edit Automaton である．このオートマトンと Insertion Automaton として表されるポリシークラスは同じであり， Suppresion Automata はそれらのサブセットであることを示した．また，セキュリティオートマトンは，プログラムがポリシーに違反したとき単に停止させるように実行列を変更しているだけであると考えれば，セキュリティオートマトンは Edit Automaton の特別な場合であると考えることができる．

P.W.L.Fong [10] は，セキュリティオートマトンとして表されるポリシーを詳細に分類することをめざしている．モニタが実行列を保持するのではなく単に実行ステップの集合を保持する機構を考え，それを Shallow Hisotry Automata (SHA) としている．この機構ではアクセス制御や一貫性に関する性質を強制することが可能である．たとえば， Chinese Wall Policy や Low Water Mark Policy （[4] 参照）である．

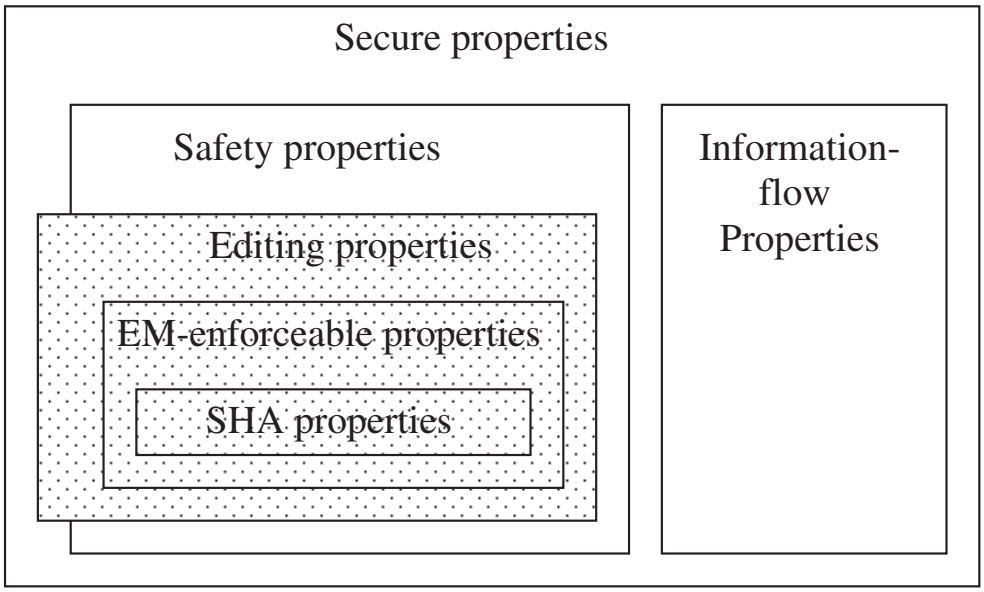

図 2.2: 強制可能なセキュリティポリシーの分類

これまでに明らかにされているポリシークラスの間の関係を 図 2.2 に示す．網掛けの部分がこれまでに各オートマトンによって強制可能であることが示されている性質である．この図に示されている情報流に関する性質は実行列の集合としては表せないことが知られている [20, 33]．セキュリティオートマトンは，実行列上の性質として表されるポリシーについて考えている．一方，情報流に関する性質は実行列の集合上の性質として表される．以降では，セキュリティオートマトンを拡張し，そのようなモニタでは情報流に関する性質も強制できることを示す.

2.3 実行監視強制可能なセキュリティポリシー

F.B.Schneider [29] の仕事にしたがって実行監視の強制機構により強制可能なセキュリティポリシーのクラスを定義する．ターゲットがセキュリティポリシー P を充たすとは，ターゲットの可能な実行列の集合を Σ として $P(\Sigma)$ が真でありまたその場合に限る，と考えることにする．そして，モニタによってポリシーを強制するためには，そのポリシーは以下に述べる条件を充たしていなければならない．モニタは単一の実行列だけを監視するので，強制可能なポリシーは単一の実行列に関する計算可能な論理式として定義されなけらばならない．さらに，実行監視の定

義より，実行列の未来からの情報を利用することはできない．つまり，ある実行列 σ が与えられたとして，任意の $i \in N$ について，もし $\hat{\mathrm{P}}(\sigma[..i])$ が偽であるならば，その後，どのような実行列にたいしてもまた偽でなければならない．これをつぎのように表す．

$$\forall i.\ \neg\hat{\mathrm{P}}(\sigma[..i]) \Rightarrow \forall \tau \in \Sigma\ \neg\hat{\mathrm{P}}(\sigma[..i] \cdot \tau) \tag{2.1}$$

また，同様にモニタによって不受理になる実行列は必ず有限の列で不受理にならなければならない．このことをつぎのように表す．

$$\forall \sigma \in \Sigma.\ \neg\hat{\mathrm{P}}(\sigma) \Rightarrow \exists i.\ \neg\hat{\mathrm{P}}(\sigma[..i]). \tag{2.2}$$

(2.1)，(2.2) から強制可能なセキュリティポリシーは， prefix-closed でなければならない．さらに，B.Alpern [1] らによると，安全性を受理する Büchi オートマトン A は，そのすべての状態を受理状態とした Büchi オートマトン $cl(A)$ と受理する言語が等しい．すなわち，受理言語 $L(A)$ として $L(A) = L(cl(A))$ である．$L(cl(SA))$ はあきらかに prefix-closed である．よって，ポリシーをすべての状態を受理状態とするオートマトンとし，その受理器をモニタと考えることにする．このとき，ポリシーが強制可能であることを以下のように定義する．

定義 2.1 (強制可能性) セキュリティポリシー $\hat{P}$ が強制可能であるとは，そのポリシーが *prefix-closed* であること，すなわち，ある σ が与えられたとして

$$\forall i.\hat{P}(\sigma) \Rightarrow \hat{P}(\sigma[..i])$$

のときである．

2.4　セキュリティオートマトンの拡張

付加構造を導入するために Schneider のセキュリティオートマトンをつぎのように拡張する，まず，セキュリティオートマトンを以下のように定義する．有限状態オートマトン (Q, Q_0, I, δ) であり，ここで，それぞれ

- Q は状態の集合とする，
- Q_0 は初期状態の集合で， $Q_0 \subseteq Q$ であるとする，
- I は入力記号の集合であり，

- $\delta : I \times Q \to 2^Q$ は遷移関数,

とする．つぎに，各状態はターゲットの実行列の集合によってラベル付けされるものとする．各状態のラベルは，ラベルの集合 L としてラベル付け関数 $\gamma : I \times L \to L$ によって決定される．

ターゲットの実行列を $s_0\, s_1 \cdots \in \Sigma$，そのときのアクション列を $a_1\, a_2 \cdots$ とする．オートマトンは Q_0 ではじまり，そして，入力記号 $I_i = (a_i, s_i) \in A \times S$ に依存して，現在の状態 $Q' \subseteq Q$ をつぎの状態へ遷移させる．つぎの状態は，$\cup_{q \in Q'} \delta(q, I_i)$ によって決定される．遷移関数 δ は，$A \times S$ と L 上の述語 p_{ij} によって与えられる．つぎの状態を以下のように表すことにする．$l_i \in L$ として,

$$\{q_j | q_i \in Q' \wedge p_{ij}((a_i, s_i), l_i)\}. \tag{2.3}$$

とする[1]．もし $q_i \in Q'$ で p_{ij} が真であるならば，その入力記号 (a_i, s_i) がオートマトンによって受理され，そうでないときには拒否される．このように表されたオートマトンによって受理される言語は prefix-closed であることはあきらかである．よってこのオートマトンによって表されるポリシーは強制可能である．

2.5 情報流に関する性質

情報流に関する性質を定義するためにマルチレベルシステムといくつかの演算を定義することからはじめる．

2.5.1 マルチレベルシステム

マルチレベルシステムとは，プログラムやデータがいくつかの階層にわかれているシステムのことである．この階層を**セキュリティレベル**という．状態マシン (S, A, Σ, s_0) はシステムを定義する一つの方法である．S は，状態変数の集合 V からある値の集合への関数であり，A はシステムの実行ステップを抽象化したアクションの集合であり，Σ は状態の有限あるいは無限の列の集合であり，そして s_0 は初期状態である．可能な実行列は，システムの状態の列とみなすことができ，各実行列は s_0 では

[1] もし，この強制機構の内部でアクセス制御も行なうのであれば (2.3) にそれを表す述語 p'_{ij} を加えればよい.

じまる．ある単一の $a \in A$ による遷移は $s_i \xrightarrow{a} s_j$ と表すことにする．システムは s_i でアクション a を生じ，状態 s_i からつぎの状態 s_j に遷移したことを意味する．さらに，この遷移の概念をアクションの列へ拡張する．$\alpha = a_1, a_2, \cdots, a_n$ として，そのとき，

$$s_0 \xrightarrow{\alpha} s_n.$$

は，

$$s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots \xrightarrow{a_n} s_n.$$

を意味する．この遷移は推移的である．すなわち，

$$s_0 \xrightarrow{a_1\ a_2 \cdots a_m} s_m \text{ and } s_m \xrightarrow{a_{m+1} \cdots a_n} s_n.$$

ならば

$$s_0 \xrightarrow{\sigma} s_n$$

である．

ここで，上述の状態マシンを拡張する．セキュリティレベルの集合 D を導入する．これは，ユーザあるいはデータのグループの集合である．ユーザやデータはあらかじめ決められたグループに属しているものとする．また，D への関数 dom:$A \to D$ が定義されているとする．a をあるターゲットが実行したアクションであるとしたとき，dom(a) はそのターゲットのオーナのセキュリティレベルを表している．また，D の上には半順序関係 $<$ が定義される．もし $d_1 < d_2$ であるならば，d_2 から d_1 へは情報流は存在しないことを意味する．$d_1 < d_2$ のような表明の集合が**情報流ポリシー**である．

定義 2.2 (マルチレベルシステム) セキュリティレベルの集合 D が与えられたとして，状態マシンを (S, A, Σ, s_0, D) と拡張する．またこのとき，アクションは入力アクションと出力アクションとに区別される．それぞれの集合を I, O とする． I と O は互いに疎である．

2.5.2　情報の等価性

まず情報とは何かを考える．ターゲットは，情報をいくつかの変数に保持している．この変数を状態変数として，この値が情報である．この

値は，ターゲットの実行履歴に依存して決まる値である．そこで，状態変数の集合 V とある時刻 $T \in \mathsf{N}$ が与えられたとして，ある状態変数の時刻 0 から T までの履歴を以下のように定義する．それは，可能な実行列の集合を定義域とする関数であり，h_T と表すことにする．

定義 2.3 (履歴) $\sigma \in \Sigma$ を可能な実行列として，この関数は，以下のような長さ $T+1$ の有限列を導く．

1. $h_T(\sigma)$ の *i* 番目の要素 s_i は， V の要素である各変数へのそのときのアクション a_i による代入であり，
2. 各変数 $v \in \mathsf{V}$ について， $h_T(\sigma)$ の *i* 番目の要素によって v に代入される値は σ の i 番目の要素によって代入される値と同じである．

情報流に関して議論するとき情報とはなにかをまず考えなければならないが，この履歴の $T+1$ 番目の状態が時刻 T における情報であるとする．また， T における状態を同様に h_T で表すことにする．

前述のように情報を定義したので，あるユーザからみたときの状態の等価性を以下のように定義する．

定義 2.4 (状態の等価性) あるユーザのセキュリティレベル $d \in D$ ，二つの状態 s_i, s_j が与えられたとして，$s_i =_d s_j$ であるためには，そのユーザが観測できる変数 v として，すべての v について

$$s_i(v) = s_j(v)$$

となることである．ここで i, j は自然数である．

$s_i =_d s_j$ であるとき，そのユーザがそれぞれの状態から得られる情報は同じであるということにする．

また，この等価性を利用して二つの実行列の等価関係を以下のように定義する．Π は実行列の履歴の集合とする．

定義 2.5 (実行列の等価性) $\alpha, \beta \in \Pi$ として，あるセキュリティレベルのユーザが実行するどんな アクション a についても，もし

$$s_0 \overset{\alpha\cdot\langle a\rangle}{\rightarrow} s_i, s_0 \overset{\beta\cdot\langle a\rangle}{\rightarrow} s_j$$

であるとき

$$s_i =_{dom(a)} s_j,$$

であるならば

$$\alpha =_{dom(a)} \beta$$

である．ここで，$\langle a \rangle$ は，単一のアクションの実行列とし，$\cdot$ は列の連結を表しているとする．

セキュリティレベル $dom(a)$ があきらかなときにはそれを省略する．

システムが情報漏洩がおこらないという意味で安全であることを定義するために，実行列上に制限演算 $\uparrow$ を導入する．

定義 2.6 (制限演算) $d \in D, \alpha \in \Sigma$ がそれぞれ与えられたとして，A_d を以下のように定義し，

$$A_d \equiv \{a \in \alpha | d \in D \ \wedge \ dom(a) < d\}$$

そして $\alpha \uparrow A_d$ を A_d に含まれるアクションだけをその順序で取り出した α の部分列とする．α が空であれば $\alpha \uparrow A_d$ もまた空である．

$\uparrow$ は推移的であると仮定する．ゆえに，$\langle a \rangle$ は単一のアクションの列，$\alpha = \alpha' \cdot \langle a \rangle$ として，

$$\alpha \uparrow A_d = \alpha' \uparrow A_d \cdot \langle a \rangle \uparrow A_d.$$

となる．

2.5.3　安全なシステム

システムが情報流について安全であるとは，直観的には，システムの振舞を観測し得られる結果が特定のユーザのグループに制限されたシステムの振舞から得られる結果と同じであれば，そのシステムは安全である，ということである．もし，それらの結果が異なるときには，そのシステムには情報流ポリシーに反した情報流が存在する．これまでいくつかの情報流に関する性質が提案されている [13, 31, 25, 18, 26, 28]．O'Halloran [25] の Noninferece を形式的に定義し，これらの性質を用いて情報漏洩がおこらないという意味で安全なシステムを定義する．ここでは，A.Zakinthinos [33] らの定義にしたがって述べる．

情報流に関する性質はシステムの入出力に関する安全な関係を定義している．その性質は，直観的には，システムのある実行列 $\sigma \in \Sigma$ のパージもまたそのシステムの可能な実行列であるならば安全なシステムであるということである．A.Zakinthinos らの定義にしたがって Low Level Equivalent Set (LLES) を導入する．$\alpha \in \Pi$ とする．あるセキュリティレベルより低いユーザが実行するアクションの集合 L が与えられたとして集合 $LLES_L(\alpha, \Pi)$ を以下のように定義する．

$$LLES_L(\alpha, \Pi) = \{\beta \mid \alpha \uparrow L = \beta \uparrow L \wedge \beta \in \Pi\}$$

この定義を用いて O'Halloran の Non-inference を以下のように定義する．その考え方は，実行列からすべての入出力アクションを削除したアクション列もまた実行可能な列でなければならないということである．

定義 2.7 (Non-inference 安全) システムの履歴の集合 Π として，

$\forall \alpha \in \Pi.$
NONINFERENCE$(LLES_L(\alpha, \Pi))$
where
NONINFERENCE$(A) \equiv \exists \tau \in A.\tau \uparrow H = \langle\rangle$.

ここで H は $A - L$ となるようなアクションの集合で，高レベルアクション集合と呼ぶことにする．高レベル出力アクションが高レベル入力アクションが出現したとき以外には表れず，出力アクションによってただ一つの状態に遷移するときすなわち，システムが決定的であるとき Non-inference は Gogen と Meseguer [13] の不干渉性と等価である．

さらに，McLean [20] は，高レベル出力アクションが高レベル入力アクションによらず出現すると仮定して一般化 Non-inference を提案した．これを以下のように定義する．

定義 2.8 (一般化 Non-inference 安全) システムの可能な実行列の集合 Σ として，

$\forall \alpha \in \Pi.$
GN$(LLES_L(\alpha, \Pi))$
where
GN$(A) \equiv \exists \beta \in A.\beta \uparrow (H \cap I) = \langle\rangle$.

我々のモニタで安全であることを保証できるかどうかはこの 2 つの性質の間でわかれる．

2.6　コバートチャネルの発見

この節では，情報流ポリシーが強制可能であることを示す．本稿で用いる不干渉性をあらたに定義し，拡張されたモニタではこの不干渉性のもとで強制可能であることを示す．F.B.Schneider のセキュリティポリシーのクラスはアクセス制御に制限される[2]. 情報流に関する性質は実行列の集合の集合上の性質として定義され，実行列の集合上の性質としては定義できない [20, 33]．よって，情報流ポリシーを単純なモニタを利用して強制することはできない．しかし，拡張されたモニタではあらたに定義した不干渉性のもとで情報流ポリシーを強制できる．

2.6.1　本稿で利用する情報流に関する性質

本稿で利用する情報流に関する性質について述べる．この定義の基本的アイデアは unwinding technique [14] を利用することである．直感的には，情報流に関する性質は，実行列の集合上の性質として表される．一方で， unwinding technique を利用することで，その性質を各実行ステップに関する性質と考えることができる．

不干渉性を構成する助けとしていくつかの関数を定義する．X は状態の集合， a はアクション， α は有限のアクション列とし，以下の関数を定義する．

$$\begin{aligned} \text{reachable}(X, \alpha) &\equiv \cup_{s \in X} \{t \mid s \xrightarrow{\alpha} t\}. \\ \text{next}(X, a) &\equiv \cup_{s \in X} \{t \mid s \xrightarrow{a} t\}. \end{aligned}$$

そして，

$$\begin{aligned} &\text{reachable}(X, \alpha \cdot \langle a \rangle) \\ &\quad = \text{next}(\text{reachable}(X, \alpha), a). \end{aligned}$$

という関係が成り立つと仮定する．不干渉性は，これらの関数を利用して以下のように定義される．

定義 2.9 (不干渉安全) 情報流ポリシー r とシステム (S, A, Σ, s_0, D) が与えられたとして，そのシステムがポリシー r に関して不干渉安全であるとは，すべての $a \in A, \sigma \in \Sigma, i \in \mathsf{N}$ について

[2][29] 内の脚注 6 では，情報流を制限しかつ safety property であるポリシーは存在すると彼はいっている．彼のクラスはアクセス制御行列のようなアクセス制御ポリシーのクラスのことであるように思われる．もし，この解釈がただしいならば，そのようなポリシーを強制してもコバートチャネルは存在しうる．

$$\begin{aligned}&\text{next}(\text{reachable}(\{s_0\}, \sigma[..i], a)\\ &= \text{next}(\text{reachable}(\{s_0\}, \sigma[..i] \uparrow A_{\text{dom}(a)}), a).\end{aligned}$$

であることとする．

モニタは，ある単一のアクションを受理するか否かを決定するためにそのアクションによる遷移を調べるので，そのアクションによる安全な通信を定義するばよい．各ターゲットの状態をあわせた状態を $s \in S$ として，そしてあるセキュリティレベル d のどんなアクション $a \in A_d$ についても $\langle b \rangle \uparrow A_{\text{dom}(a)} = \langle\rangle$ であるアクション $b \in A$ を考え，もしあるシステムが情報流ポリシー r に関して不干渉安全であるならば，そのとき

$$\text{next}(\text{reachable}(\{s\}, \langle b \rangle), a) = \text{next}(\{s\}, a)$$

となる．ここで，$\langle\rangle$ は空列である．いま，集合 $\text{reachables}_a(s)$ を以下のように定義する．

$$\begin{aligned}&\text{reachables}_a(s) \equiv\\ &\quad \{\text{reachable}(\{s\}, \langle b \rangle) \mid \langle b \rangle \uparrow A_{\text{dom}(a)} = \langle\rangle\}.\end{aligned}$$

$s_b \in \text{reachables}_b(X)$ として，もし $\text{dom}(a) < \text{dom}(b)$ であるならば，つぎの不変表明が，単一のアクションについて得られる．s_b のすべての状態は $\text{dom}(a)$ から見て s に等しい．$t \in s_b$ とし，

$$s =_{\text{dom}(a)} t$$

である．直観的には，t は b を実行したときのシステム状態を表し，s は b を実行しなかったときのシステム状態を表す．つまり，$\sigma \uparrow A_{\text{dom}(a)} \in \Sigma$ であり，その時刻の状態 $MI(\sigma, d)$ ここで $d = \text{dom}(a)$ とあわしたとき，

$$MI(\alpha, d) =_d t \tag{2.4}$$

であり，(2.4) から，

$$\begin{aligned}&\forall t' \in \text{next}(\{t\}, b). \exists s' \in \text{next}(\{MI(\sigma, d)\}, b).\\ &\quad MI(\sigma, d) =_d t \Rightarrow s' =_d t'.\end{aligned} \tag{2.5}$$

をえる．この場合では，$\text{dom}(b)$ は d についてローカルに不干渉である．

つぎに，もし，この表明が任意の実行列でいつも真であるならば，システムは与えられた情報流ポリシーについて不干渉安全であることを示す．

補題 2.1 r は情報流ポリシー，(S, A, Σ, s_0, D) をシステムとすると，任意の $i \in N, \sigma \in \Sigma, a, b \in A, d \in D$ について，もし式 *(2.5)* が真であれば，そのシステムは r に関して不干渉安全である．

証明 2.1 i に関する帰納法を用いる．

Basis $i = 0$ のとき，$\uparrow$ の定義より

$$\begin{aligned}&\text{next}(\{\sigma[..0]\}, \langle\rangle)\\ &\quad = \text{next}(\{\sigma[..0]\} \uparrow A_d, \langle\rangle)\\ &\quad = \{s_0\}.\end{aligned}$$

ゆえに情報漏洩は存在しない．

Inductive Step i について，補題 *2.1* が正しいと仮定する．$X \equiv \text{reachable}(\{s_0\}, \sigma[..i])$ とし，$Y \equiv \text{reachable}(\{s_0\}, (\sigma \uparrow A_d)[..i])$ として，そのとき，任意の $a, b \in A$ について，

> もし式 *(2.5)* が正しいならば，
> $\text{next}(\text{next}(X, a), b) = \text{next}(\text{next}(Y, a), b)$ であり，そのとき，システムは不干渉安全である．

ことを示す．

Case 1: $d < \text{dom}(b) \in r$ とする．
帰納法の仮定より，

$$\text{next}(X, a) = \text{next}(Y, a). \tag{2.6}$$

となる．$d < \text{dom}(b)$ と式 *(2.6)* から

$$\begin{aligned}&\text{next}(\text{next}(X, a), b)\\ &\quad = \quad \text{next}(\text{next}(Y, a).b)\\ &\quad = \quad \text{reachable}(\{s_0\}, \sigma[..i] \cdot \langle ab \rangle).\end{aligned}$$

である．

Case 2: $d < \mathrm{dom}(b) \notin r$ とする．
$d < \mathrm{dom}(b)$ によって，$\mathrm{next}(\mathrm{next}(Y, a), b) = \mathrm{next}(Y, a)$ となる．帰納法の仮定より，

$$\mathrm{next}(X, a) = \mathrm{next}(Y, a).$$

いま，

$$\mathrm{next}(\mathrm{next}(X, a), b) = \mathrm{next}(X, a).$$

であることを示す．式 *(2.5)* が正しいと仮定して，

$$\mathrm{next}(\mathrm{next}(X, a), b) \subset \mathrm{next}(X, a)$$

を示す．もし， b が式 *(2.3)* の p_{ij} を充たし， b 安全なアクションであり，式 *(2.5)* が正しいならば，そのとき，強制機構の仮定から任意の $s \in \mathrm{next}(A, b)$ にたいして，$s =_d t$ である $t \in \mathrm{next}(\mathrm{next}(X, a), b)$ が存在し，そのとき，

$$\mathrm{next}(X, a) \subset \mathrm{next}(\mathrm{next}(X, a), b).$$

である．よって，

$$\mathrm{next}(\mathrm{next}(X, a), b) = \mathrm{next}(X, a).$$

となり，ゆえに，補題 *2.1* は任意の実行列について正しい． ■

補題 2.1 は，もし (2.5) がいつも真であるならば，そのとき，システムが情報流ポリシー r に関して不干渉安全であることを示している．このとき，不干渉安全なシステムは Non-inference を充たしていることはあきらかである．ある σ の各状態について考える．不干渉であれば必ず $\sigma[..i] \uparrow A_{\mathrm{dom}(a)}$ が存在する．このとき， $A_{\mathrm{dom}(a)} \cap H$ は空集合となり，$\sigma[..i] \uparrow A_{\mathrm{dom}(a)} \uparrow H$ は空列となるので Non-inference の条件を充たす．しかし，一般化 Non-inference は充たさない．$\sigma[..i] \uparrow A_{\mathrm{dom}(a)}$ が存在するが，これは高レベルの入出力アクションすべてを削除したものであるからである．また，補題 2.1 の逆は，その不変表明が導出された過程からあきらかにただしい．

2.6.2　情報流に関する性質の強制

我々は，ある与えられた情報流ポリシーのもとでシステムにコバートチャネルが表れたことを発見することに興味があるので，情報流に関する性質が拡張された強制機構で強制可能であることを見ていく．実行列の集合 L によってセキュリティオートマトンの状態をラベル付けする拡張を行なった．そこで，あるセキュリティレベル d として $\sigma \uparrow A_d$ を各セキュリティレベルごとに考え，この集合を L とする．α を時刻 T までに観測した有限の実行列として，式 (2.5) より，式 (2.3) の p_{ij} をつぎのようにする．

$$p_{ij} \equiv \forall d.\ h_T(l_i \cdot \langle a \rangle \uparrow A_d) =_{\mathrm{dom}(a)} h_T(\alpha \cdot \langle a \rangle) \tag{2.7}$$

このとき，つぎの状態のラベル l' は

$$l' = \gamma(l, a) = l \cdot \langle a \rangle \uparrow A_d$$

となる．

定理 2.1 式 *(2.7)* で定義された遷移関数をもつオートマトンは強制可能であり，観測された実行列は与えられた情報流ポリシー r について不干渉安全である．

証明 2.2 補題 *2.1* から，式 *(2.5)* がある実行列でいつも正しいという条件はシステムの観測された実行列が情報流ポリシーについて安全であるということの十分条件である．また，補題 *2.1* の逆は，その不変表明が導出された過程からあきらかにただしい．ゆえに，システムは情報流ポリシー r について不干渉安全である．

無限実行列 $\sigma \equiv s_0\ s_1\ \cdots\ s_T\ s_{T+1}\ \cdots \in \Sigma$ が与えられたとして，どんな時刻 T においても，s_T が存在し，それに対応するエミュレータの状態任意の $a \in A$ にたいして，$t_T = MI(\sigma, d)$ が存在し，s_T から s_{T+1} に a によってシステム状態が遷移したならば，s_{T+1} は a を実行しないときの状態と $\mathrm{dom}(a)$ のレベルから見たとき等しい，すなわち $s_{T+1} =_{\mathrm{dom}(a)} t_{T+1}$ であるとき，その $\sigma[..T+1]$ を受理するオートマトンを考える．明らかに任意の時刻における履歴によって到達可能な状態は受理状態であり，このオートマトンはすべての状態が受理状態である．よって，このオートマトンを SA としたとき $cl(SA)$ は同じ言語を受理する．また，補題 *2.2* より，$h_T(\sigma[..T] \uparrow A_d)$ は $\sigma[..T]$ だけから計算することができ，$\sigma[..T]$ は実行可能である．したがって，不干渉性は強制可能である．　■

つぎに，$h_T(\sigma[..T] \uparrow A_d)$ が $\sigma[..T]$ が観測した実行列だけから計算することができることを示す．

補題 2.2 あるシステム (A, S, Σ, s_0, D) として，すべての $\sigma \in \Sigma$ にたいして，σ が実行可能ならば，$\sigma \uparrow d$ は実行可能な列でそれは σ だけから計算することができる．

証明 2.3 まず，$\sigma[..T]$ とする． T に関する帰納法を用いる．

Basis $T = 0$ のとき，$\sigma[..T]$ は空列で，$\uparrow$ の定義より $\sigma[..T] \uparrow A_d$ もまた空列である．すべてのシステムは空列を実行可能であるのとしているので，明らかに $\sigma[..T] \uparrow A_d$ そのシステムで実行可能である．$\sigma[..T] \uparrow A_d$ は $\sigma[..T]$ だけから計算可能である．

Inductive Step $T = n$ として，補題 *2.2* が正しいと仮定し，$\sigma \cdot \langle a \rangle$ が実行可能であるとする．ここで， $a \in A$ である．

Case 1: 情報流ポリシー r，あるセキュリティレベル d，$d < \mathrm{dom}(a) \in r$ として，そのとき，$\uparrow$ の定義より，

$$\sigma[..T] \uparrow A_d = \sigma \cdot \langle a \rangle.$$

よって，$\sigma \cdot \langle a \rangle$ が実行可能であれば $\sigma[..T] \uparrow A_d$ もまた実行可能である．

Case 2: $d < \mathrm{dom}(a) \notin r$ のとき，$\uparrow$ の定義より

$$\sigma[..T] \uparrow A_d = \sigma.$$

帰納法の仮定から，$\sigma[..T] \cdot \langle a \rangle \uparrow A_d$ も実行可能である．また，$\uparrow$ の定義より，$\sigma[..T] \cdot \langle a \rangle \uparrow A_d$ は $\sigma[..T]$ だけから計算することができる．以上より，補題 *2.2* 任意の実行列で正しい． ■

以上のことより，不干渉安全は強制可能であり，不干渉安全なシステムは Noninference を充たすのでこれも強制可能である．一方，一般化 Noninference は強制できない．なぜなら，高レベル入力だけをパージしたときエミュレータが正しく模倣できないかも知れないからである．

2.7　二つのセキュリティレベルを持つシステムの場合

この節では，二つのセキュリティレベルをもつシステムについて考える．そのセキュリティレベルをそれぞれ $high, low$ として，$low < high$ とする．いま，それぞれのレベルのプロセスを $\mathcal{L} \equiv (S_L, L, \Sigma, s_0)$ と $\mathcal{H} \equiv (S_H, H, \Sigma', s'_0)$ とし，この二つを合成したシステムを考え，その状態を $\langle s_i^{\mathcal{L}}, s_i^{\mathcal{H}} \rangle$ とする．ここで，$s_i^{\mathcal{L}}$ は $\mathcal{L}$ の状態を表し，$s_i^{\mathcal{H}}$ は $\mathcal{H}$の状態を表すとする．システムの実行列すなわちアクションの列は，$\sigma_L \in \Sigma$ と $\sigma_H \in \Sigma'$ をインターリーブしたものになる．システムの状態遷移は以下のいずれか一方になる．

$$\begin{aligned} \langle s_i^{\mathcal{L}}, s_i^{\mathcal{H}} \rangle &\xrightarrow{a} \langle s_{i+1}^{\mathcal{L}}, s_i^{\mathcal{H}} \rangle \quad \text{if } a \in L \text{ and } a \notin H, \\ \langle s_i^{\mathcal{L}}, s_i^{\mathcal{H}} \rangle &\xrightarrow{a} \langle s_i^{\mathcal{L}}, s_{i+1}^{\mathcal{H}} \rangle \quad \text{if } a \notin L \text{ and } a \in H. \end{aligned}$$

このシステムは， input-total であるとする．input-total とはすべての状態でどんなアクションも実行可能であるということである．もし，この仮定をしないときには， $\mathcal{L}$はアクションの実行が許されたかどうかによってなんらかの情報を予測できてしまう．

以降では，二つのレベルをもつシステムのコバートチャネルを発見する様子を示す．その強制機構は，先に述べたように，付加構造としてエミュレータをもち，アクセス制御のためのポリシーと情報流ポリシーが与えられる．ここでは，二つのレベルだけであるので簡単に，情報流ポリシーは $low < high$ となる．モニタとエミュレータの状態遷移を図 2.3 に示す．遷移は遷移グラフとして表されている．エミュレータは，low レベルのプロセス上で実行されるアクション列だけによるシステムの状態遷移を模倣し，モニタの状態は q0 と q1 によってラベル付けされている．q0 はそのオートマトンの初期状態を表す．その遷移グラフの枝は $H, L, NotLeak$ によってラベル付けされている．H, L は，それぞれの構成要素の状態集合を表しおり，ある情報へのアクセスが許されることを表している．$NotLeak$ は，$\mathcal{L}$が観測できる情報は自身のシステム状態だけであると仮定し，エミュレータの状態を s_e として， $s_e =_{\mathrm{dom}(l)} s_i^{\mathcal{L}}$ であるとき真となる $\mathcal{L}$の状態の集合であるとする．ここで $l \in L$ とする．l が実行されたときにはいつでも $NotLeak$ が真であるかどうかをこの強制機構は検査する．式 2.3 との対応は，$NotLeak$ が，2.6 節で述べた情報流ポリシーを強制するための条件を表している．情報流ポリシーを $\{low < high\}$ とし，$NotLeak$ を

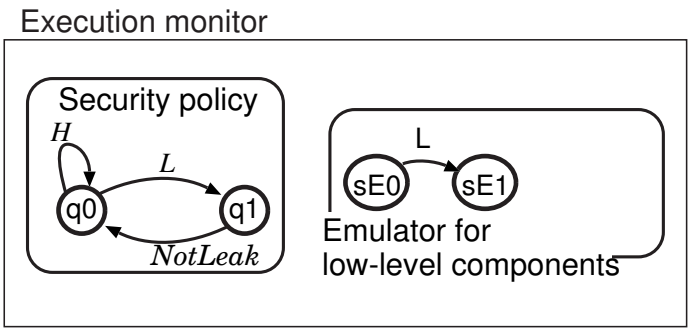

図 2.3: 情報流ポリシーの強制

$s_i^{\mathcal{L}} = s_j^{\mathcal{L}}$ と定義する．$\langle s_0^{\mathcal{L}}, s_0^{\mathcal{H}} \rangle$ は初期状態，T はある時刻として，可能であるどんな σ についても，このシステムは $\langle s_i^{\mathcal{L}}, s_i^{\mathcal{H}} \rangle$ と $\langle s_j^{\mathcal{L}}, s_j^{\mathcal{H}} \rangle$ が存在し，そのとき

$$\langle s_0^{\mathcal{L}}, s_0^{\mathcal{H}} \rangle \xrightarrow{\sigma[..T]} \langle s_i^{\mathcal{L}}, s_i^{\mathcal{H}} \rangle$$
$$\langle s_0^{\mathcal{L}}, s_0^{\mathcal{H}} \rangle \xrightarrow{\sigma[..T]\uparrow L} \langle s_j^{\mathcal{L}}, s_j^{\mathcal{H}} \rangle.$$

である．いま，定理 2.2 を証明してこのシステムが，情報流ポリシー $\{low < high\}$ に関して不干渉安全であることを示す.

定理 2.2 任意のアクション $a \in H \cup L$ として，a によって $s_i^{\mathcal{L}}$ から $s_j^{\mathcal{L}}$ へ状態遷移したとすると，もし，いつも $s_i^{\mathcal{L}} =_{low} s_j^{\mathcal{L}}$ であるならば，このシステムは $\mathcal{H}$ から $\mathcal{L}$ への情報流は存在しない.

証明 2.4 T に関する帰納法を用いる．$\sigma[..T]$ を $\alpha \cdot \langle a \rangle$ として，

Basis $T = 0$ のとき，$\langle s_0^{\mathcal{L}}, s_0^{\mathcal{H}} \rangle =_{low} \langle s_0^{\mathcal{L}}, s_0^{\mathcal{H}} \rangle$ は明らかに正しい.

Inductive Step $T > 0$ のとき,

Case 1: $a \in L, a \notin H$ とすると，そのとき，$\mathcal{L}$ は $s_j^{\mathcal{L}}$ と $s_i^{\mathcal{L}}$ だけを観測できるので，$s_j^{\mathcal{L}} =_{low} s_i^{\mathcal{L}}$ とすると，そのとき $\langle s_i^{\mathcal{L}}, s_i^{\mathcal{H}} \rangle =_{low} \langle s_i^{\mathcal{L}}, s_j^{\mathcal{H}} \rangle$ であり，定義 2.9 より，あきらかにこのシステムは不干渉安全である.
$s_j^{\mathcal{L}} \neq_{\mathcal{L}} s_i^{\mathcal{L}}$ のときには，実行監視により強制的に停止され，情報漏洩を防ぐ.

Case 2:　$\alpha \notin L, \alpha \in H$ とすると，システム状態の遷移の定義より，$s_i^{\mathcal{L}} = s_i^{\mathcal{L}}$ となる．よって，$\langle s_i^{\mathcal{L}}, s_i^{\mathcal{H}} \rangle =_{low} \langle s_i^{\mathcal{L}}, s_j^{\mathcal{H}} \rangle$ となり，定義 2.9 よりあきらかに与えられた情報流ポリシーに関して不干渉安全である．

以上よりいつも $s_i^{\mathcal{L}} =_{low} s_j^{\mathcal{L}}$ であるならば，このシステムは，情報漏洩は存在しない．

この例ではセキュリティレベルは二つであるとした．しかし，実際のシステムではもっと多くのレベルを持っていると思われるので，レベル数に比例した数だけエミュレータが必要になる．エミュレータをレベル数分だけ用意することはそれだけで資源を消費してしまい，実用的でない．

2.8　我々の強制機構の概要

図 2.4 に示すように，情報流は情報流制御とアクセス制御によって制御される．情報流制御は，エミュレータの状態と実際の計算資源の状態を比較し，もしそれらが異なるときには，観測したアクションを実行したプロセスを停止することで行なわれる．これは，コバートチャネルの受信者を停止することに等しい．

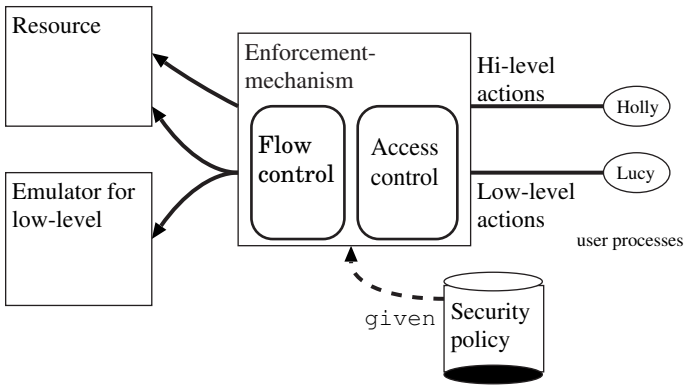

図 2.4: 強制の仕組み

つぎに，我々の強制機構に与えるポリシーの記述言語について述べる．そのサンプルを図 2.5 に示す．アクション ”`abort`” と ”`allow`” は特別なアクションであり，`abort` は，ターゲットシステムを強制的に停止させることを意味し，`allow` は，アクションを実行し，その結果があれば，それをターゲットに返すことを許し，同時にポリシーをつぎの状態に遷移

させることを意味する．これらのアクションはいつでも実行可能である．アクション ”nis” もまた特別なアクションである．しかしながら，このアクションは，いつでも遷移可能なわけではない．図 2.4 に示されている情報流制御がエミュレータの結果と実際の計算資源へ同じアクションを実行したときの結果を比較し，その値がパラメータにバインドされとときだけ遷移可能である．

(bind remove_l ”syscall:lucy:sys_unlink”)

は，アクション `remove_l` をプロセス `lucy` によって呼び出されたシステムコール

`sys_unlink`

に対応させることを意味する．これ以外の明示的に対応づけられていないシステムコールは無視される．

(dominate Holly Lucy)

は情報が Holly から Lucy に流れないことを表明している．このような表明の集合によって表される情報流ポリシーを強制するための条件は，明示しない．それは，情報流制御内に埋め込まれている．

(read_h(f):(if $\cdots$))

は，`read_f` に対応づけられたシステムコールを観測したならばつぎの状態に遷移し，さらにそのシステムコールの引数を `f` にバインドすることを意味している．演算子 ”++” は，非決定的な選択実行を意味している．そして演算子 ”||” は二つのプロセスの合成を意味している．最後に, 演算子 ”define” 抽象的なプロセスを宣言するための特別な演算子である．”Start_policy” は，ポリシーの初期状態であり，固定されている．たとえば，Lucy が呼び出したシステムコール `sys_unlink` を強制機構が観測し，ポリシーはつぎのような遷移を生じるとき,

$$\text{Start\_policy} \xrightarrow{\text{remove\_l}} \text{(if (f=”}\cdots\text{”)}\cdots)$$

もし，計算資源とエミュレータそれぞれからの結果が同じを得るならば, そのとき，強制機構は allow を生じてつぎの状態に遷移する．

```
(bind read_l "syscall:lucy:sys_read")
(bind write_l "syscall:lucy:sys_write")
(bind remove_l "syscall:lucy:sys_unlink")
;; the following is an information-flow policy.
(dominate Holly Lucy)
;; the following is access policies.
(define NI ()
  (nis(x):
    (if (x=true)
        (~allow:Start_policy)
        (~abort:Start_policy))))
(define Holly ()
  ((read_h(f):
    (if (f = "conf.txt")
        (~allow:Start_policy)
        (~abort:Start_policy)))++
   (write_h(f,d):
    (if ((f = "conf.txt")||
         (f = "covertchannel.txt"))
        (~allow:Start_policy)
        (~abort:Start_policy)))++
   (create_h(f):~allow:Start_policy)++
   (remove_h(f):~allow:Start_policy)))
(define Lucy ()
  ((create_l(f,d):
    (if (f = "covertchannel.txt")
        (~allow:NI)
        (~abort:Start_policy)))++
   (remove_l(f):
    (if (f = "covertchannel.txt")
        (~allow:NI)
        (~abort:Start_policy)))++
;; the following process is an initial process.
(define Start_policy ()
  (Rights_of_holly++Right_of_lucy))
```

図 2.5: コバートチャネルを発見するためのポリシーの例

2.9　おわりに

我々は，F.B.Shcneider のセキュリティオートマトンを，その各状態をアクション列の集合でラベル付けするように拡張し，そのようなオートマトンで表されるモニタによって部分的に情報流に関する性質を強制できることを示した．これを図 2.6 に示す．Nondeducibility や一般化 noninfterference は強制することはできない．なぜなら，それらの性質は単にパージするだけでなく適当なアクションを挿入あるいは削除したアクション列を考えなければならないからである．Non-inference は偶然，エミュレータからの情報と一致してしまいコバートチャネルを発見できないことがある．しかし，1.3 節で述べたように Nondeducibility や一般化 noninterference ではパージするだけでなくいくつかのアクションを挿入あるいは削除した列

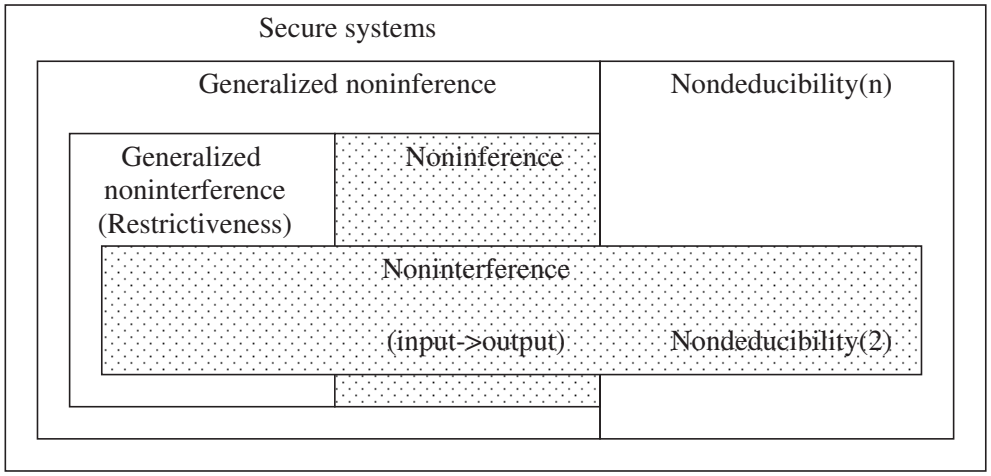

図 2.6: 強制可能な性質の分類

を考えることでこの問題を回避している．よって，本来は Nondeducibility や一般化 noninterference がより安全であり，これらの性質を強制できることが望ましい．

我々の拡張モニタは，各セキュリティレベルごとにエミュレートしなければならない．システムが n 個のセキュリティレベルをもつとき，n−1 個のエミュレータが必要である．このことは，モニタを実現するためには多くの計算資源を費さなければならないことを意味し， n=2 のとき以外はあまり現実的ではない．

また，状態がアクション列の集合でラベル付けされたオートマトンを考えた．本稿では情報流に関する性質を表すためにラベルを観測したアクション列の部分列の集合としたが，ラベルを適切に定義することでさらに他の性質を表すこともできる．今後は，そのようなときの性質の特徴をあきらかにする必要がある．たとえば， Schnieder は観測した実行列と現実の実行順序は同じであると仮定しているが，この仮定を除いたときどのようなポリシーが強制可能であるかを議論することができると思われる．観測した実行列の各アクションを任意の順序で並べ替えた実行列の集合をラベルとして議論することで特徴があきらかにすることを試みる．

第3章　システムモデルと GDB を用いたソースレベルモデル検証

3.1　はじめに

安全なシステムを構成するためには，動的な検査とあわせて静的な検査を用いるべきであると考える．そこで，我々は実際に実行されるコード（ソースコードと実行コード）を直接検証するツールも開発する．

また，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二つの検査工程があり，一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，もう一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．この二つの工程それぞれで検証器が開発されている．SPIN [15] のような従来の検証器ではモデルだけを扱い[1]，最近の検証器 [12, 22, 23] ではソースコードだけを扱う．また， Java PathFinder [32] は Java のソースコードを SPIN のモデル記述言語である Promela [15] に変換してソースコードを間接的に試験する．CMC [22] 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，その環境自身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

[1]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

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，ある性質を充たすことを検証する上で，その環境が正しいことが保証されてる．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 [21] 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充たしているか否かを検査する．言い替えると，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充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，ある値にたい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一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。

本章の構成を以下に述べる．3.2 節では，関連研究について述べる．3.3 節では，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ついて述べ，ソースコードを直接検証する手順を述べる．3.4 節では，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．3.5 節では，今回開発した検証器の制約について議論する．3.6 節でまとめを述べる。

3.2 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 [15] 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．

3.2.1 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 [11, 15] 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 [17]．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つくりだしている．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．

3.2.2 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 3.2.1 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 [12] 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 [22] 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 とのインタフェイ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，この環境自身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し，このモデル自身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，上記二つの検証器は環境の非決定的な振舞を模倣するために適当な範囲内の一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を定義することが可能である．

Java PathFinder [32] 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用いており，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 [23] 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．

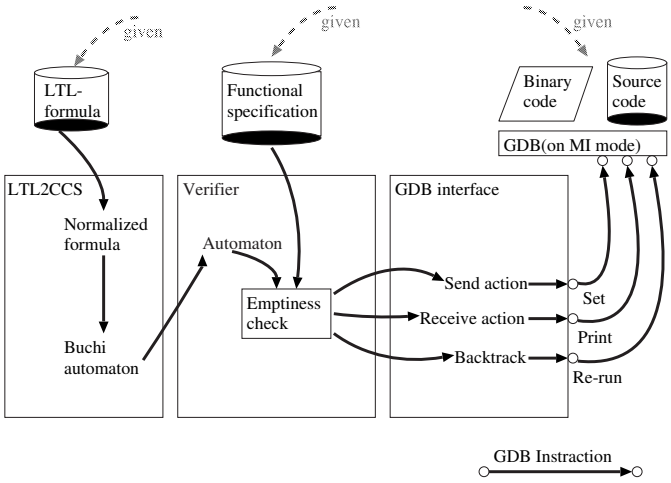

図 3.1: 検証器の構成

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，ある性質を充たすことを検証する上では，その環境の正しさが保証されている．

3.3 検証器の設計

この節では，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，つぎ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．

3.3.1 検証器の概要

我々の検証器は三つのサブシステムから構成される（図 3.1 参照）．一つはモデルを解釈実行する検証器本体であり，二つ目は GDB を制御するサブシステムであり，三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，一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 [21] にもと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 3.3.2 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（この詳細は第 3.4 節で述べる）．状態を生成しているとき，もしある状態で二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，一つの状態からは多くとも二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なう．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 3.3.3 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ひと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 rerun を利用し，バイナリコードを再実行する．ここまで，三つの構成要素の連携のしかたについて述べた．つぎに， 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もと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もっていて，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もち，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ひと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なわれる．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 3.2 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も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 3.3 参照）．

ソースコードはシステムの構成要素の一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 3.3.3 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いけないとする．も

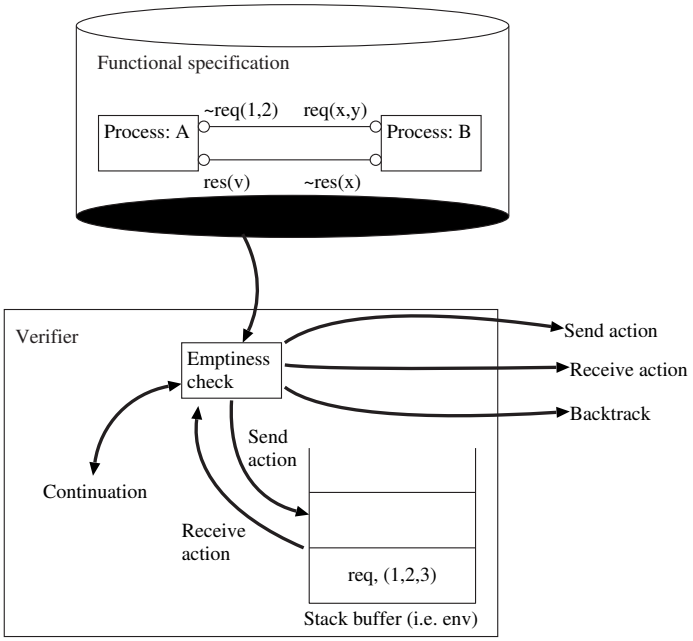

図 3.2: 検証器の概要

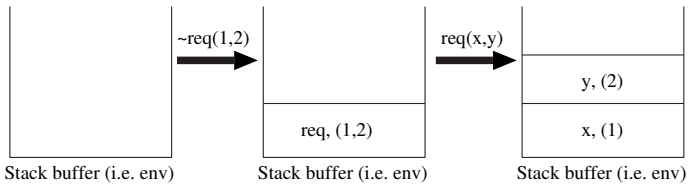

図 3.3: 通信路の様子

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．この節の残りの部分では，簡単な例を用いてソースコードを試験する過程を説明する．

3.3.2　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．

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

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，残りの二つをアクション前置という．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二種類がある．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，プロセス合成は二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 3.4 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を表しており， 28 行目から 29 行目はサーバの振舞を表している．

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 演算子を含んではいけない．この検証器では，原子論理式は集合

$$\mathsf{V} \cup \mathsf{D} \cup \{\texttt{ture}, \texttt{false}\}$$

上の関係 $<=, <, >=, >, ==, !$ を用いて表される．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用いて表現する（図 3.5 参照）．具体的には，各状態を表すプロセス式は，

```
(define State_id ()
  (if (atomic_propositons)
      (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 `atomic_propositons` 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

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

となる．また，モデルには必ず一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，$1 \leq i \leq n$ である．すなわち，モデル M として

$$\begin{aligned} M \equiv & \{(\mathsf{define}\ P_i\ (x, \cdots)\ E_i) : 1 \leq i \leq n\} \\ & \cup \{(P_i) : 1 \leq i \leq n\} \end{aligned}$$

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 $eval : \mathsf{E} \rightarrow \mathsf{E}$（付録 B 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．プロセス式の集合を以下のように定義する．

定義 3.1 (プロセス式の集合) 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．

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

$$
\begin{aligned}
&\{E||F\} \cup \{E'||F' :\ E' \in Sub(E) \\
&\quad \text{and } F' \in Sub(F)\} \\
Sub(&\texttt{if } (C)\ (E)\ (F)) = \\
&\{\texttt{if } (C)\ (E)\ (F)\} \cup Sub(E) \\
&\quad (\text{if } C = true) \\
Sub(&\texttt{if } (C)\ (E)\ (F)) = \\
&\{\texttt{if } (C)\ (E)\ (F)\} \cup Sub(F) \\
&\quad (\text{if } C = false) \\
Sub(&P(e_1, \cdots, e_n)) = \\
&\{P(x_1, \cdots, x_n)\} \cup \{Sub(A[e_i/x_i])\} \\
&\quad (\text{if } (\textsf{define } P(x, \cdots)\ A))
\end{aligned}
$$

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．以降では，この集合を E と表すときもある．

3.3.3 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いけない．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を設定する．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，

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

の形をもっていると考える．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 をもち，

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

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も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．この二つのアクションはアトミックに実行される．よって，この二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みなすことができる．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 [16]．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[2]．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，我々は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．

[2] 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．

3.3.4 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，つぎに対応するプロセスをモデルから削除する．たとえば，図 3.6 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 3.4 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` を用いて宣言している．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用いる．検証アルゴリズムについては，第 3.4 節で述べる．

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，その試験環境自身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．$Sub(\texttt{Echo\_client})$ 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みなすことができ，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．

3.4 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

全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得ないので，このように仮定する．モデルは全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．3.3.2 節で述べたように性質のオートマトンは $(\texttt{accept} : E)$ 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二つのデータ構造をもつ．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，到達可能な全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．

3.4.1　初期状態

3.3.2 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身であり，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，初期プロセス自身である．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．

3.4.2　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， B 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．これら二つの演算子がプロセスの非決定的な振舞を表現している．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二つであるので，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，つぎ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一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．ルートを印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その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 [5] と呼ばれる方法であり，この疑似コードを図 3.7 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，二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，そのまま二つのプロセス式の字句が全て同じであるということである[3]．よって，

[3]我々のモデル記述言語は名前空間が一つだけであるので，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．

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．

つぎに，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 3.3.3 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，その値に対する一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として表される．たとえば，ソースコード中の変数 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．

3.4.3　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，3.3.2 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，$env : \mathsf{V} \rightarrow \mathsf{D}$ 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，3.3.2 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つぎの遷移を判断する．

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つぎ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．これらは，言い替えるとモデルから観測されるプログラムの振舞だけを考え，それによるモデルへの影響を考慮して，モデルだけで検証したときに充たしていた性質をまだ充たしているか否かを調べている．

3.5 ソースコード検証器の制約

この節では，今回開発した検証器の制約について議論する．最初の二つは，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いけないとする．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おこなうコードは，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を

$$\text{alloc}(1) : E\text{++}\text{alloc}(0) : E$$

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おこなうとき，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．

我々の検証器は，その構成からプログラムとモデルの等価性を検査しているわけではない．モデルの一部すなわちシステムの機能の一部をプログラムに置き換えても，そのプログラムがモデルに悪影響を与えない，ということを調べているだけである．

3.6　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，一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，ソースコードは一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，3.3.3 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用いて場合をわければよい．

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 [29] 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 [29] をを拡張し，可能な範囲で予測したり，付加情報をえることにより 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 [2]

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．これまで明らかであることについて以下に述べる．我々の検証器はその構成から，状態空間に利用するメモリ空間はコンパイル時に決定され一定量である．プロセス式はリストで表現されるが，このリストの各セルは 24 バイト[4]でありこの整数倍のメモリ量となる．検証時には，必要に応じて状態を生成していくが，生成できる状態数はプロセス式の複雑さにも依存し一概にはわからない[5]．もし状態数が多いときには，このメモリ空間を使い果たすまで状態を生成する．よって，検証の終了までの時間もこのメモリ空間のサイズに依存する．また，利用している検証アルゴリズムから二つの状態の等価の定義の仕方にも依存する．現在は，プロセス式の字句が同じであれば同じ状態であるとしているが，この定義については検討する必要があると思われる．

[4]利用するコンパイラによるが，gcc 3.3.2 では 24 バイトである．

[5]ECHO サービスの例では 978 個の状態を生成し，約 0.7 秒かかる．計測には CPU: Pentium3 800MHz, メモリ: 128 M OS: Linux 2.4.31 を利用した．

```
 1: ;(bind srv_res "target:echos:10")
 2: ;; Timer
 3: (define Timer()
 4:   (start(init_v):Timer1(init_v)))
 5: (define Timer1(time)
 6:   (~tick(time-1):tick(time):
 7:     (if (time=0)
 8:         (~timeout(TRUE):STOP)
 9:         (Timer1(time)))))
10: (define Timeout()
11:   (timeout(zz):
12:     (if (zz)
13:         (~display("TIMEOUT"):~srv("quit"):STOP)
14:         (stop))))
15: ;; Echo client
16: (define Recv1()
17:   (srv_res(yyy):
18:     (Send(100,n-1))))
19: (define Recv() (Recv1++Timeout))
20: (define Send(x,n)
21:   (if (n=0)
22:       (STOP)
23:       ((~srv(x):~start(5):
24:         (Recv)))))
25: (define Echo_client()
26:   (Send(100,5)||Timer))
27: ;; Echo server
28: (define Echo_server()
29:   (srv(req):~srv_res(req):STOP))
30: ;; Initial process
31: (Echo_server || Echo_client)
32: ;(Echo_client)
```

図 3.4: ECHO サービスの例

*

```
1: (define Node0002 ()
2:   (if (! ((x) = (yyy)))
3:       (accept: Node0002)
4:       (Abort)))
5: (define Init ()
6:   (if (! ((x) = (yyy)))
7:       (accept: Node0002)
8:       (Init)))
9: (define _MC_START () (Init))
```

図 3.5: 線形時間論理式 ”◇ !(x=yyy)” の記述

*

```
 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 3.6: ECHO サーバの例（echos.c）

*

```
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 3.7: 検証アルゴリズム

第4章　ポリシーとモデルの記述言語

4.1　はじめに

この記述言語は，モデル記述とポリシー記述の両方で用いられるが，この章では，セキュリティポリシーの記述方法について述べる．悪意あるコードからホストの計算資源を保護方法がいくつかある（サンドボックス，セキュリティオートマトン）．サンドボックスでは，極端に制限されたポリシーでコードを実行することで計算資源を保護してきた．セキュリティオートマトン [29] は，イベントの時系列を考慮して制限するセキュリティポリシーを記述する．これは，コードのイベントあるいは状態を入力記号としたオートマトンとしてセキュリティポリシーを定義しなければならない．我々の実行監視はイベントの列だけでなく付加情報も利用することを想定している．ここでは，この様な複雑なセキュリティポリシーをどのように記述するべきかについて議論する．

セキュリティポリシー記述言語について，以下のことを議論する．

- より利便性を増すためには，比較的複雑なセキュリティポリシーを容易に記述できる必要がある．この際に用いる抽象化概念にどのようなものを用いるべきか．
- セキュリティポリシーは，どのようなレベル[1]のイベントも記述できなければならない．現実のイベントをどのようにセキュリティポリシーの世界で抽象化したものと対応付けるか．

本章では，4.2 節では，モデル化する際の基本的なアイデアを述べる．4.3 節では，ここで用いる記述系の構文と意味を直観的に述べている．4.5 節では簡単な記述例を示す．4.6 節ではまとめとして今後の検討課題を述べ，簡単な考察を加える．

[1]ここで云うレベルとは，仮想マシンや CPU の命令，JAVA のメソッドコールといった意味でのレベルである．言い替えると，ある命令の抽象度のことである．

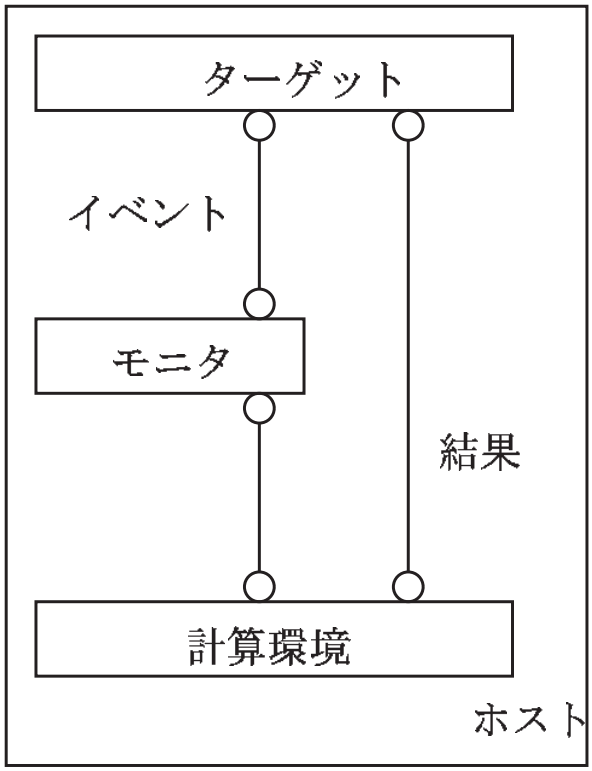

図 4.1: システムのモデル

4.2　基本的な考え方

4.2.1　セキュリティ強制のモデル

最初に，セキュリティオートマトンを用いた強制機構の一般的な機構について述べる（図 4.1 参照）．セキュリティオートマトンを実現した機構をここではモニタと呼ぶことにする．モニタを導入したシステムの基本的な動作を次のように考える．ターゲット[2]，モニタ，システムそれぞれがインタラクションをもつ．ターゲットのイベントを観測し，受理するときにはそれを計算環境に渡し，計算環境がそれを実行する．その結果は計算環境からターゲットに直接渡される，と考える．セキュリティポリシーは，ターゲットとモニタのインタラクションをモデル化したものである．モニタは実行結果には関与しない．したがって，セキュリティポ

[2]ターゲットとは悪意あるコード全般のことをいっている

リシーには実行の結果は現れない．ひとつのシステムは複数のポリシーを持つことになる．ポリシーは，それ自身ひとつのオートマトンである．それは決定的に選択実行されるものと考える．

4.2.2 抽象化概念

次に抽象化の方法について議論する．J.A.Goguen [13] らは，不干渉表明 (noninterfering assertion) の集合によってセキュリティポリシーを表している．F.B.Schneider [29] は，状態をそれを表すいくつかの状態変数の組として定義している．状態遷移はガード付きコマンドによってそれを表している．ガードの記述方法については触れていない．U.Eringsson [9] らは，ガードを実際のイベント，オペランド上の述語として表現している．我々はプロセス代数に基づいた体系を用いることでセキュリティポリシーを記述する．記述系については後の節で詳しく述べる．以下では，提案する記述系の利点について述べる．

実際のシステムではセキュリティポリシーはより複雑な構造を記述する必要がある．たとえば，２つの異なる移動コードが１つのホスト上に同時に動作しているとものする．この２つがインタラクションを持つ場合を考える．このインタラクションによってはどちらかの実行を制限しなければならない．これは [9], [29] で用いられている記述法で記述することは難しい．なぜなら，２つの主体（移動コード）の間にデータフローが存在するからである．

4.2.3 イベントとアクションの対応

ここで述べることは，セキュリティポリシーの性質を議論する上では本質的ではない．しかし，実際に，ポリシーを記述する上では，入力記号の集合を定義することは重要なことであると考えられるのでここで触れておく．

セキュリティポリシーはどのようなレベルの実行も記述できなければならない．言い替えると，現実のイベントをオートマトンの入力記号と対応付ける方法の議論である．[9] では，オペレータ，オペランドは必ず特定の変数に対応づけられる．我々は，個々の特別な変数を用意するのではなく，特別な演算子 bind を用意する．アクション名や変数名はセキュリティポリシー設計者が決定する．

4.3　ポリシー記述言語

本節では，ポリシー記述言語の構文と意味を直観的に示す．提案する言語は， CCS[21] に基づいている．現実のイベントをセキュリティポリシーの世界で抽象化するために bind と言う特別な演算子を導入する．これは

(bind action(arg,$\cdots$) ”level:event”)

と記述する． level はイベントの抽象度を表す．たとえば， Java VM の命令を表す ”JVM” ，メソッド呼びだしを表す ”method” と言ったキーワードを用いる．event は各レベル毎に指定された記述を行う．action は引数を持つことができる．引数は現実のイベントのオペランドや引数がバインドされる．たとえば， JVM のレベルであれば各バイトコードのオペランドがそれぞれバインドされる．

特別なアクションとして others と abort を持つ． others は特定のアクション以外のアクションを意味する．abort はターゲットを強制的に停止させる特別なアクションである．

アクション前置 (action prefix)，合成 (composition)，選択 (summation) の各演算子は，それぞれ : ， || ， ++ を用いて表される．このほかにプロセス定数を定義するために define と言う特別な演算子が用意されている．それぞれの意味は CCS とほぼ同様であるが，合成と選択のオペランドには，オペランド間に優先順位が定義されている．左のオペランドが優先度が高い．

複数のセキュリティポリシーがある場合には，それぞれのオートマトンが選択的に動作するものとしてモデル化すれば良い．ポリシーは，その実行を許す場合とそうでない場合どちらを記述しても良い．ただし，許されないアクションを観測したときにはその直後に abort を実行するようにしなければならない．

4.4　構文と意味の概要

CCS は並行計算モデルとも云われるように，並行に実行されるプログラムやリアクティブシステムを対象とした計算モデルである．世の中すべてのものがプロセスであり，ユーザもまたプロセスであると考える．各プロセスはシーケンシャルに動作し，それらが互いに同期をとって並行

に動作する．それぞれの計算は対応するアクションが実行することで行われる．

値の集合 値の集合 VALUE として整数，文字列，真偽値 {TRUE,FALSE} を許す．この集合上の演算は整数の演算は四則演算および不等号が，真偽値上の演算は and および or を表す '&','|' が許される．このほかに not の意味で '!' が定義されている．文字列上の演算は '+','−','/' が定義されている．演算子 + は二つの文字列を左から右の順番で連結する．ここで文字列を str とする．演算子 − は str/n となり， n は自然数である．演算結果は str の先頭の n 文字からなる文字列となる．演算子 / は str/n となり， n は自然数である．結果は str の先頭から n 文字を除いた残りの文字列となる．文字列は長さと順序と文字が等しいとき，二つの文字列は等しいとする．各演算子の関係は

$$\mathrm{str} = (\mathrm{str} - \mathrm{n}) + (\mathrm{str}/\mathrm{n})$$

でなければならない．

アクション前置 アクションの実行を表す演算子は，':' であり，

$\sim$a(v):$\cdots$
a(x):$\cdots$

と記述する．ここで v は VALUE の要素であり，x は VALUE 上の変数である．アクションはいくつかの値を持つことが可能である．その値はただ一度だけ評価される．アクションの集合 NAME は

$$\{\mathrm{Letter}\}\{\mathrm{Letter}|\mathrm{Digit}\}^*$$

によって表される最小の集合である．アクションには出力と入力の区別がある．ここで NAME の要素 a とすると，出力アクションを '$\sim$a' と表し，入力アクションは単に 'a' と表す．CCS とは異なり $\sim\sim$a = a とはならない．また内部アクションは特に定義されていない．実行可能なアクションは次のように決定される．出力ポートは常に遷移可能であり，入力ポートは受信データがすべてバインドされたときだけ遷移可能である．複数の遷移が可能であるときには，より左側に出現するアクションによって状態遷移する．

選択　アクションの選択実行を表すための演算子が用意されている．二つのプロセス A,B とすると選択は

A ++ B

と表される．いずれか可能なプロセスだけが状態遷移を行い，それ以降は選択されたプロセスだけのアクションが可能なときに実行される． A,B どちらも遷移可能なときにはより左側のプロセスの遷移が優先される．

合成　システムはいくつかの構成要素から構成される．これを得るために合成演算子 '||' が定義されている．二つのプロセス A,B とすると合成は

A || B

と表される．いずれか可能なプロセスだけが状態遷移を行う．同じ名前を持つ入力アクションと出力アクションが対応する．出力アクションが変数を持つときには対応する入力アクションが持つ値がその変数にバインドされる．

ガードつき実行　ガードつき実行を表すために if という演算子を定義している．これは

(if cond Then Else)

と記述する．cond はブール式であり， Then と Else はそれぞれあるプロセス式である． cond が真のときには Then だけが実行され，偽のときには Else だけが実行される．

プロセス定数　これまで述べてきたものの他に define という演算子が用意されている．define はプロセス定数を定義するための演算子である．

(define proc (vars) Body)

と記述する．プロセスは引数を持つことが可能である．Body はプロセス式である．

BIND 演算子 特別な演算子 bind はアクションを，ここでは ターゲットの適当な代入文に対応付けるための演算子である．

(bind name address)

と記述する． address は，target:filename:lineno と指定しなければならない． target はキーワードであり，モデル記述のときに用いる．filename はファイル名を指定し， lineno はそのファイル内の対応させるべき代入文の行番号を指定する．この他に syscall や method があり，それぞれポリシーを記述するときに用い，システムコールや Jave のメソッド呼出しに対応する．また， TCP/IP を用いてプロセスが通信するときには IP アドレス，ポート番号を指定するために server および client といったキーワードも用意されている．

特別なアクションとプロセス 特別なアクションとして display と key がある．これらは，ディスプレイとキーボードをそれぞれひとつのプロセスと考え，それらとのコミュニケーションを取るためのアクションである．したがって一方向の通信しか行えない．key はとくに rigid variable をユーザが指定するために用いることを意図している．

accept という特別なアクションは性質を表すオートマトンの受理状態を表すための特別な言語である．モデル記述時にこれを用いたときの振る舞いは未定義である．

ユーザは明示的にプロセスの停止を記述必要があるとき，そのための特別なプロセス stop を利用することができる．このプロセスにはつぎのような性質がある．

$$\text{stop} = \text{stop} || \text{stop} = \text{stop}{+}{+}\text{stop}.$$

全てのプロセスが停止状態にあるか，それ以上遷移することができないときプロセスは停止する．

このほかに Abort という特別なプロセスが用意されている．性質を表すオートマトンがそれ以上遷移しないことを明示的に表すときに用いる．モデル記述においてこれを用いたときは，その振る舞いは未定義である．ポリシー記述において用いる abort とは区別される．

名前空間と演算子の結合規則 各変数，各アクションのスコープはダイナミックスコープを採用している．名前空間は一つである．構文を BNF

を用いて定義したものを付録 A に示す．また，各演算子の形式的な意味を付録 B に示す．

各演算子の結合規則や優先順位は表 4.1 に示す．優先順位は表の上にある演算子ほど高くなる．define, bind は入れ子にはできない．これら，演

表 4.1: 演算子の優先順位と結合則

演算子	結合則
define, bind, if	
++, ||	左結合
[],{}	左結合
:,~	右結合
|, &	左結合
<,>,=,,<=,>= &	左結合
+,− &	左結合
∗,/,%	左結合
!	右結合

算子はキーワードであり予約語である．accept, abort は大文字と小文字は区別される．stop = STOP である．

4.5 記述例

4.5.1 ファイルアクセス制御

あるファイルにアクセスし，その内容を読み込み，送信または移動する場合を考える．このとき，ファイルによってはアクセス後の移動やそれの送信を禁止しなければならない．たとえば，あるファイルはそのホスト上での閲覧は許すが，それを他へ持ち出すことは禁止すると言った場合である．この例では価格表がそのようなファイルであるとしている．このようなポリシーをオートマトンで表すと図 4.2 となる．このようなオートマトンを図 4.3 のように記述する．

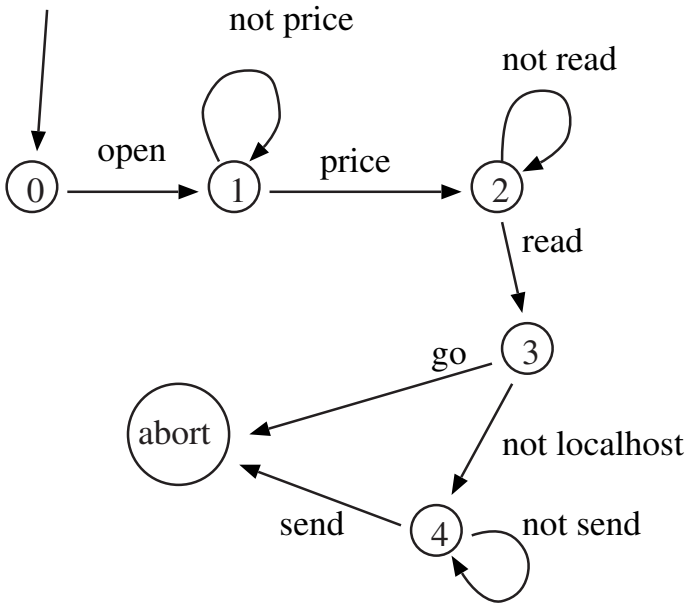

図 4.2: 状態を持ったファイルアクセス制御の例

4.5.2 二つの主体が協調する場合の例

これまで単一のターゲットについて述べてきた．ここでは２つのターゲットが協調する場合について考える．4.5.1 節と同様なポリシーを考える．ただし，本節では２つの主体がそのファイル（価格表）を授受しているものと仮定する．このとき，そのファイルを受け取ったターゲットの実行にも受信後の移動を禁止するといった制限が加えられる必要がある．このポリシーの記述を図 4.4 に示す．ここではオートマトンだけを示す．

4.6 結論と今後の課題

これまでポリシー記述系について述べてきた．それは [29] で示されたポリシーのクラスは十分に表現できる．さらに，２つのターゲットがデータをやりとりしながら並行に動作する場合のポリシー記述の必要性を例を用いて示した．ガードつきコマンドだけでは難しかったこの記述を，我々の記述系では可能にしている．

また，これまで本章では各アクションが持つ値の型や表現については詳しく述べていない．それぞれのレベルに適した型やその表現を個別に

```
;;; イベントとアクションの対応
(bind open(path)
  "method:/CLASS/file")
(bind read()
  "method:/CLASS/read")
(bind send(des)
  "method:/CLASS/write")
(bind go(des)
  "method:/CLASS/go")
;;; 状態遷移
(define FileOpen ()
  (if (path="/shop/price")
      (FileOpen)
      (FileRead)))
(define FileRead ()
  ((read():HasRead)
   ++(others():FileRead)))
(define HasRead ()
  ((send(des):
     (if (des="localhost")
         (Start)
         (abort)))
  ++((go(des):abort)
  ++(others():HasRead))))
(define Start ()
  (open(path):FileOpen))
;;; オートマトンの開始
(Start)
```

図 4.3: 状態を持ったファイルアクセス制御のセキュリティポリシー

```
(define Send ()
  ((msgAB(path):Send)
  ++(others():Send)))
(define IsRead ()
  (if (path="/shop/price")
      (read(file):Send)
      (others():IsRead)))
(define PrincipalA ()
  (open(path):
     (if (path="/etc/passwd")
         (abort)
         (IsRead))))
(define RecvMsg ()
  (go(des):abort ++ others():RecvMsg))
(define PrincipalB ()
  ((msgAB(arg):RecvMsg)
  ++(others():PrincipalB)))
(define Start ()
  (PrincipalA || PrincipalB))
```

図 4.4: 協調する場合のセキュリティポリシー

用意しかなればならない．それは，実装時に議論するべきであると思われる．我々は，この記述系のインタプリタを開発したが，これを JVM などの実際のシステムに導入したいと思う．インタプリタは，ポリシー自身の振舞いを検査するための道具として利用することができる．記述されたポリシーがセキュリティ上の制約を満たしていることは明らかでない．したがって，セキュリティポリシーを検証する必要がある．

第5章　全体のまとめ

5.1　本研究の成果

本研究では，情報流ポリシーが付加的な構造を持つ実行監視によって，本研究のために定義した意味で強制可能であることを示した．我々の最初の目的は，付加情報（たとえば，モデル解析や型理論）を利用したセキュリティポリシーの特徴を明らかにすることであった．情報流ポリシーは，この立場からの最初のアプローチである．

また，これら複雑になるであろうセキュリティポリシーを記述するための適当な抽象化の方法を本研究では提案した．それは，プロセス代数にもとづいた概念を持っている．線形時間論理から変換されるオートマトンを表現するには十分であり，より一般的なシステムの振舞モデルを記述することができる．このことは，ソースコードレベル検証器を開発したことでも確認されている．

モデル検証器の開発は，実行監視により強制できない性質をソフトウェアの構成時に検証する目的で開発した．そのために，より実際に実行されるコードに近い実行コードを GDB 上で実行することにより検証するツールを開発した．

5.2　未解決の問題

実行監視がイベント列だけでなく，実行監視の外側から何らかの方法（たとえば，モデル解析や型理論）で得られる情報もいっしょに利用するとき，より多くのポリシーが強制可能であると思われる．このことから，どのような情報を利用したとき，どのようなセキュリティポリシーが強制できるか，利用する付加情報の観点から特徴づける必要がある．

静的に検証するとき，状態数を抑制するために partial order reduction や data abstraction などの手法が利用される．このとき，特定の性質は検証できない．たとえば， partial order reduction を用いたときには，線形時間

論理の Next 演算子は検証することができない．このような静的な解析で検証できない性質を動的な解析により強制できることが望ましい．しかし，このような性質が強制可能であるかどうかは明らかにすることができなかった．

もうひとつ未解決の問題は，実行監視が観測するイベントの順序が実際に観測した順序と同じであると仮定しているが，この仮定をしないときに強制可能であるポリシーの特徴について議論していない．ネットワークを経由して観測する防火壁（ファイアーウォール）などでは明らかに順序を仮定することはできない場合がある．単一のホスト上でも同様な問題が発生する．イベントを観測するためにプログラムを書き換えたとする．観測するためのコードは必ずしもアトミックではない．複数のプロセスが強調して動作しているとしたとき，OS によってプロセスを切替えられるタイミングによっては，観測した順番と異なる順序でイベントが実行される可能性がある．以上のことを考慮してもう一度セキュリティポリシーを特徴づける必要があると思われる．

謝辞

本研究を進めるにあたり，終始御指導いただいた東京工業大学情報理工学研究科計算工学専攻助教授 渡部卓雄博士に深く感謝致します．また，審査に御参加下さいました東京工業大学情報理工学研究科計算工学専攻教授 米崎直樹博士，東京工業大学情報理工学研究科計算工学専攻教授 佐伯元司博士，東京工業大学情報理工学研究科計算工学専攻助教授 権藤克彦博士，東京工業大学情報理工学研究科計算工学専攻助教授 西崎真也博士にもお礼申し上げます．さらに，多くのご助言を頂いた東京工業大学情報理工学研究科計算工学専攻助手 萩原茂樹博士および渡部研究室のみなさまにも感謝致します．

ソースレベル検証器は IPA 未踏プロジェクトの支援を受けて開発しました．関係者の方に深謝致します．とくに，検証器の開発に協力して頂いた Michael Reinsch 氏と Sushil Shrestha 氏に感謝致します．

最後に，両親・祖母（永藤滋・昌子・秋子）のこれまでのこと，これからのことに感謝致します．

参考文献

[1] B. Alpern and F. B. Schneider. Recognizing safety and liveness. In *Distributd Computing*, pages 117–126. Springer-Velag, 1987.

[2] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. Technical Report TR-649-02, Princeton University, June 2002.

[3] D. Bell and L. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE, March 1976.

[4] M. Bishop. *Computer seurity:Art and Science*. Addison-Wesley, 2003.

[5] E. M. Clarke,Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[6] Department of Defense. Trusted Computer System Evaluation Criteria, 1985.

[7] Department of Defense. A guide to understanding covert channel analysis of trusted systems, 1993.

[8] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*, pages 87–95. ACM Press, 2000.

[9] U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. *ACM*, pages 87–95, 2000.

[10] P. W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2004.

[11] Formal Systems(Europe) Ltd. Failures-Divergences Refinement: User manual, 1994.

[12] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, pages 476–479, 1997.

[13] J. A. Goguen and J. Meseguer. Security policies and security models. In *In Proceedings of the 1982 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

[14] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–86, 1984.

[15] G. J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison Wesley, September 2003.

[16] L. Lamport and F. B. Schneider. The “hoare logic” of CSP, and all that. *Programming Languages and Systems*, 6(2):281–296, 1984.

[17] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 147–166. Springer-Verlag, Berlin Germany, 1996. 1055.

[18] D. McCullough. A hookup theorem for multilevel secruity. *IEEE Transactions on Software Engineering*, 16(6):563–568, 1990.

[19] J. McHugh. Covert channel analysis: A chapter of the handbook for the computer security certification of trusted systems. Technical memorundum 5540:080a, Naval Research Laboratory, Washington D.C., 1995.

[20] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[21] R. Milner. *Communication and concurrency*. Printice-Hall, 1989.

[22] M. Musuvathi, D. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of*

the Fifth Symposium on Operating Systems Design and Implementation, december 2002.

[23] Y. Nakagawa, R. Potter, M. Yamamoto, M. Hagiya, and K. Kato. Model Checking of Multi-Process Applications Using SBUML and GDB. In *The Internatinal Confernce on Dependable Systems and Networks*, pages 215–220, June 2005.

[24] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. *LNCS 1419*, pages 61–91, 1998.

[25] C. O'Halloran. A calculus of information flow. In *Proceedings of the European Symposium on Research in Computer Security*, 1990.

[26] S. Pinsky and E. Zieglar. Noninterference equations for nondeterministic systems. In *14th IEEE Computer Security Foundation Workshp*, pages 3–14, 2001.

[27] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report SRI-CSL-92-02, SRI International Computer Science Laboratory, Dec. 1992.

[28] P. Ryan and S. Schneider. Process algebra and non-interference. In *In PCSFW: Proc. of The 12th Computer Security Foundations Workshop*. IEEE Computer Society, 1999.

[29] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[30] R. Sekar, C. Ramkrishnan, I. Ramamkrishnan, and S. Smolka. Model-Carrying Code (MCC): A new paradigm for Mobile-Code security. In *New Security Paradigms Workshop (NSPW'01)*, New Mexico, 2001.

[31] D. Sutherland. A model of information. In *9th National Computer Security Coference*, 1986.

[32] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *15th IEEE International Conference on Automated Software Engineering*, pages 3–11, 2000.

[33] A. Zakinthinos and E. S. Lee. A general theory of security properties. In *Proceedings of the 18th IEEE Computer Society Symposium on Research in Security and Privacy*, 1997.

付 録A　RCCS Syntax

```
Agent_Exp     ::= ( define ID ( ID_Seq ) Agent_Exp )
                |  ( define ID ( ) Agent_Exp )
                |  ( bind ID Strings )
                |  ( globalvar IDs Strings )
                |  ( if ( B_exp ) Agent_Exp Agent_Exp )
                |  ( A_Binary_Exp )
                |  (  )
A_Binary_Exp  ::= A_Binary_Exp ++ A_Label_Exp
                |  A_Binary_Exp || A_Label_Exp
                |  A_Label_Exp
A_Label_Exp   ::= A_Label_Exp [ ID_Seq ]
                |  A_Label_Exp { Relabel_Seq }
                |  A_Unary_Exp
A_Unary_Exp   ::= ID ( Value_Seq ) : A_Unary_Exp
                |  ID : A_Unary_Exp
                |  ~ ID ( Value_Seq ) : A_Unary_Exp
                |  ~ ID : A_Unary_Exp
                |  ID ( Value_Seq )
                |  ID
                |  ( Agent_Exp )
ID_Seq        ::= ID_Seq , ID
                |  ID_Seq , ~ ID
                |  ID
                |  ~ ID
Value_Seq     ::= Value_Seq , B_Exp
                |  B_Exp
Relabel_Seq   ::= Relabel_Seq  , ID / ID
                |  Relabel_Seq  , ~ ID / ~ ID
                |  ID / ID
                |  ~ ID / ~ ID
B_Exp         ::= B_Exp | C_Exp
                |  B_Exp & C_Exp
                |  C_Exp
C_Exp         ::= C_Exp < V_Exp
                |  C_Exp <= V_Exp
                |  C_Exp > V_Exp
                |  C_Exp >= V_Exp
                |  C_Exp = V_Exp
                |  V_Exp
```

```
V_Exp         ::= V_Exp + V_Term
                |  V_Exp - V_Term
                |  V_Term
V_Term        ::= V_Term * V_Unary_Exp
                |  V_Term / V_Unary_Exp
                |  V_Term % V_Unary_Exp
                |  V_Unary_Exp
V_Unary_Exp   ::= ! V_Unary_Exp
                |  Fact
Fact          ::= Iconst | Strings | TRUE | FALSE | ID
                |  ( B_Exp )
```

付 録B　RCCS Operatinal Semantics

各プロセス演算子の操作的意味を定義するために、次のような Push-Down Automata

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

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。Act はラベルの集合とし、$\rightarrow \subseteq Q \times Act \times Q$ とする。$(q_1, a, q_2) \in \rightarrow$ のとき $q_1 \stackrel{a}{\rightarrow} q_2$ 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 pair 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 pair 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つぎのように定義する。$push(a; v, s)$ 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クから、最初にあらわれた a を含む pair を削除し、その後のリストを返す。

$$\frac{}{[\sim a(v) : E, s] \overset{\sim a}{\to} [E, \text{push}(a; v.s)]}\ \text{Act}_1 \qquad \frac{\text{member(a,s)}}{[a(x) : E, s] \overset{a}{\to} [E, \text{delete}(a, s)]}\ \text{Act}_2$$

$$\frac{[E, s] \overset{\alpha}{\to} [E', s')]}{[E{+}{+}F, s] \overset{\alpha}{\to} [E', s')]}\ \text{Sum}_1 \qquad \frac{[F, s] \overset{\alpha}{\to} [F', s')]}{[E{+}{+}F, s] \overset{\alpha}{\to} [F', s')]}\ \text{Sum}_2$$

$$\frac{[E, s] \overset{\alpha}{\to} [E', s']}{[E||F, s] \overset{\sim\alpha}{\to} [E'||F, s']}\ \text{Com}_1 \qquad \frac{[F, s] \overset{\alpha}{\to} [F', s']}{[E||F, s] \overset{\sim\alpha}{\to} [E||F', s']}\ \text{Com}_2$$

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

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

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

$$\frac{[E, s] \overset{\alpha}{\to} [E', s'] \quad \text{eval}(B)}{[\text{if } B\ E\ F, s] \overset{\alpha}{\to} [E', s']}\ \text{If}_1 \qquad \frac{[F, s] \overset{\alpha}{\to} [F', s'] \quad \neg\text{eval}(B)}{[\text{if } B\ E\ F, s] \overset{\alpha}{\to} [F', s']}\ \text{If}_2$$

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

$$\frac{[E, s] \overset{\alpha}{\to} [E', s'] \quad \alpha'/\alpha \in R}{[E\{R\}, s] \overset{\alpha'}{\to} [E'\{R\}, s']}\ \text{Rel} \qquad \text{where R=\{a'/a,}\sim\text{a'/}\sim\text{a,}\cdots\}$$

$$\frac{}{[\text{define } A\ P, s] \to [\text{STOP}, s]}\ \text{Def}$$

$$\frac{}{[\text{bind } A\ string, s] \to [\text{STOP}, s]}\ \text{Bind}$$

$$\frac{}{[\text{STOP}, s] \not\to}\ \text{Stop}_1$$

$$\frac{[\text{STOP}, s] \not\to \quad [\text{STOP}, s] \not\to}{[\text{STOP}||\text{STOP}, s] \not\to}\ \text{Stop}_2$$

$$\frac{[P, s] \overset{\alpha}{\to} [P', s']}{[\text{STOP}{+}{+}P, s] \overset{\alpha}{\to} [P', s']}\ \text{Stop}_3 \qquad \frac{[\text{STOP}, s] \not\to \quad [\text{STOP}, s] \not\to}{[\text{STOP}{+}{+}\text{STOP}, s] \not\to}\ \text{Stop}_4$$

付 録C　LTL Syntax

```
Start             ::= StateFormula

StateFormula      ::= StateFormula /\ PathFormula
                  |   StateFormula \/ PathFormula
                  |   StateFormula -> PathFormula
                  |   ! StateFormula
                  |   PathFormula

PathFormula       ::= <> PathFormula
                  |   [] PathFormula
                  |   PathFormula U StateFormula
                  |   X PathFormula
                  |   Proposition

Proposition       ::= Proposition &  Atom
                  |   Proposition |  Atom
                  |   Proposition -> Atom
                  |   ! Proposition
                  |   Atom

Atom              ::= Atom =  Exp
                  |   Atom <  Exp
                  |   Atom >  Exp
                  |   Atom <= Exp
                  |   Atom >= Exp
                  |   Boolean
                  |   Exp

Exp               ::= Exp + Term
                  |   Exp - Term
                  |   Term

Term              ::= Digits
                  |   Id
                  |   ( StateFormula )

Boolean           ::= tt
                  |   ff
```

```
Action          ::= Id
                |   ~ Id
```

付 録D　LTL Semantics

我々のモデリング言語上での LTL formula の意味を与える，Formula F の意味を $[F]$ と表すことにする，Formula の意味は状態の列の集合上で定義される，すなわち，$[F]$ はその列上の boolean-value function である，状態の列を $\sigma = \langle s_0, s_1, \cdots \rangle$ として，$\sigma[F]$ はその関数が σ について適用されたときのその関数の値を表すものとする，We say that σ satisfy $[F]$ iff $\sigma[F]$ equals true. $[F]$ が σ で true であるための必要十分条件は，最初に状態とその次の状態が action F であることである，形式的には $langles_0, s_1, \cdots\rangle[F] \equiv s_0[F]s_1$ である，さらに，$s_n[F]s_{n+1}$ が true であるための必要十分条件は，$F[s_n(v)/v]$ が true となることである，ここで，$F[s_n(v)/v]$ は subsitution を表す，一般には $F[s_n(v)/v][s_{n+1}(v)/v']$ となるが，我々のモデリング言語では name space はひとつであるので $F[s_n(v)/v]$ となる，

状態とは変数の集合 Var から値の集合 Val への mapping function である，よって，変数の意味 $[x]$ は値 $s(x)$ となる，State function とは変数と定数値から構成される式である，たとえば，$x^2 + y - 3$ がそれである，ある状態 s での値はつぎのようになる，

$$s[x^2 + y - 3] = s[v]^{s[2]} + s[y] - s[3]$$

左辺の $2, 3$ は定数シンボルであり，その意味ではない，しかし，ここではその意味はシンボルが表す定数そのものであるとして $[2] = 2$ と考えることにする，

さて，状態についてもう少し考えることのする，我々のモデリング言語では状態はプロセス式そのものである，プロセス式がどのように遷移するかは環境 Env によって決まる，Env もまたプロセスの振る舞いを制御するプロセスである，よって，我々はモデルの状態を，プロセス P として $P\|Env$ と考えることにする，さらに，我々のモデリング言語では，プロセス間の通信路はスタックを仮定している，したがって，ここで考える configuration はスタックの状態と $P\|Env$ の状態の pair $[P\|Env, s]$ であるとする，いま，プロセス $P \stackrel{\text{def}}{=} \sim a(x+3) : A\|a(y) : B$, formula $F \stackrel{\text{def}}{=} y = x + 3$ を仮定する，このとき，P における a の実行は F を充たす，すなわち，

$$[\sim a(x+3) : A\|a(y) : B, s]\ [y = x + 3]\ [A\|B, s']$$

となり，このとき $s[y = x + 3]s'$ は true である，これは，直観的には $a(v), \sim a(e)$ と云う実行を $v = e$ と置き換えただけのことである，$P\|Env$ は必ず閉じたプロセス式となるので $\|$ についてだけ考えれば十分である，

各演算子の意味を以下のように定義する．Let $\sigma = \langle s_0, s_1, s_2, \cdots \rangle$.

$$
\begin{array}{rcl}
\sigma[F] & \stackrel{\text{def}}{=} & s_0[F]s_1 \\
\sigma[!F] & \stackrel{\text{def}}{=} & \neg\sigma[F] \\
\sigma[F/\backslash G] & \stackrel{\text{def}}{=} & \sigma[F] \text{ and } \sigma[G] \\
\sigma[F\backslash/G] & \stackrel{\text{def}}{=} & \sigma[F] \text{ or } \sigma[G] \\
\sigma[F \rightarrow G] & \equiv & \neg\sigma[F] \text{ or } \sigma[G] \\
\sigma[[]F] & \stackrel{\text{def}}{=} & \forall\, n \in \text{Nat} : \langle s_n, s_{n+1}, \cdots \rangle[F] \equiv \forall\, n \in \text{Nat} : s_n[F]s_{n+1} \\
\sigma[<>F] & \stackrel{\text{def}}{=} & \exists\, n \in \text{Nat} : \langle s_n, s_{n+1}, \cdots \rangle[F] \equiv \exists\, n \in \text{Nat} : s_n[F]s_{n+1} \\
\sigma[\text{X } F] & \stackrel{\text{def}}{=} & \langle s_1, s_2, \cdots \rangle[F] \\
\sigma[F \text{ U } G] & \stackrel{\text{def}}{=} & \exists\, n \geq 0 : \langle s_n, s_{n+1}, \cdots \rangle[G] \text{ and } \forall\, m : (0 \leq m < n) \\
 & & \text{and } \langle s_m, s_{m+1}, \cdots \rangle[F]
\end{array}
$$

我々のツールでは LTL formula を次のようにプロセス式に変換する，formula は全て if の条件部として表され， then 部， else 部に条件部の評価結果に対応した遷移を表すプロセス式が入る，

Example D.1 $[](y = x + 3) \stackrel{ltl2ccs}{\Rightarrow}$

```
(define A ()
  (if (y=x+3)
      (accept:A)
      (Stop)))
```

次の例は liveness property の簡単な例を表している，あるプロセスがいつかはクリティカルセクションに入れること表している，

Example D.2 $\diamond\,(sem = 1) \stackrel{ltl2ccs}{\Rightarrow}$

```
(define A ()
  (if (sem=1)
      (accept:B)
      (A)))
(define B ()
  (others:B))
```

研究業績

1. 永藤直行, 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, 情報処理学会論文誌 プログラミング, Vol.47,No.SIG11(PRO30), pp.1–12, 2006.
2. Naoyuki Nagatou and Takuo Watanabe, Run Time Detection of Covert Channels, The First International Conference on Availability, Reliability and Security, IEEE Press, pp.577–584, April, 2006.
3. Naoyuki Nagatou and Takuo Watanabe, Execution Monitoring and Information Flow Properties, DSN 2005 International Workshop on Dependable Software Tools and Methods, IEEE Press, pp.221–227, June, 2005.
4. Takuo Watanabe, Kiyoshi Yamada and Naoyuki Nagatou, Speccifying Context-Aware Runtime Security Policies using an Algebraic Policy Specification, IASTED International Conference on Software Engineering(SE 2004), ACTA Press, pp.662–667, February, 2004.
5. Takuo Watanabe, Kiyoshi Yamada and Naoyuki Nagatou, Towards a Specification Scheme for Context-Aware Security Policies for Networked Appliances, IEEE Workshop on Software Technologies for Future Embedded Systems, IEEE Press, pp.65–68, May, 2003.
6. 永藤直行, 渡部卓雄, 不干渉性の強制について, コンピュータセキュリティ, 情報処理学会 2006-CSEC-34, pp.207–214, July, 2006.
7. 永藤直行, 渡部卓雄, プログラム解析から得られる情報を用いた実行監視について, 日本ソフトウェア科学会第 21 回大会, 6A-1(5pages), September, 2004.
8. 永藤直行, 渡部卓雄, 移動コードのための機密性強制, 第 1 回ディペンダブルシステムワークショップ (DSW'04), 日本ソフトウェア科学会, pp.121–130, February, 2004.
9. 渡部卓雄, 山田聖, 永藤直行, コード書換えによるセキュリティポリシーの実行時強制機構, 日本ソフトウェア科学会第 19 回大会, 4F-2(7pages), September, 2002.
10. 永藤直行, 渡部卓雄, 移動コードの安全な実行のためのポリシー記述, 日本ソフトウェア科学会第 18 回大会, 7C-3, September, 2001.
11. 永藤直行, 組込みソフトウェアのためのテスティングツールの開発, 2004 年度未踏ソフトウェア事業, IPA, 2004.