計算モデル特論

国立情報学研究所

佐藤一郎

E-mail: ichiro@nii.ac.jp

プロセス論理

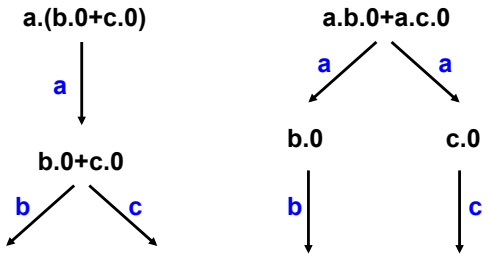
- Hennessy-Milner論理
  - 構文:
    P ::= true | false | P ∧ Q | ¬P | <a> | [a]
- 非形式的意味
  - true　真を表す
  - false　偽を表す
  - P ∧ Q　PかつQ
  - ¬P　Pの否定
  - <a>　アクションaに関する可能様相演算子
  - [a]　アクションaに関する必然様相演算子

プロセス論理の充足関係

- プロセス論理式によるプロセス表現
  - P |= F とはプロセスPは性質をFを満足（充足）
- 充足関係|=　(|= ⊆ P×F)は以下のように定義される
  - P |= <α>F とは ∃P':P–α→P'　かつ P' |= F
  - P |= ¬F とは P ⊭ F （つまりP|=Fとなることがない）
  - P |= F∧G とは P |= F かつ P |= G

プロセス論理の例

- 論理学手法を利用したプロセスの仕様記述と証明



a.(b.0+c.0) |= <a>(<b>true ∧<c>true)
a.b.0+a.c.0 |= <a><b>true ∧ <a><c>true
a.b.0+a.c.0 ⊭ <a>(<b>true ∧<c>true)

プロセス論理の例

a.(b.0+c.0) |= <a>(<b>true ∧<c>true)

a.b.0+a.c.0 |= <a><b>true ∧ <a><c>true

a.b.0+a.c.0 |≠ <a>(<b>true ∧<c>true)

検証

- **検証**とは、事実を確かめること（法律用語的には）
- ソフトウェアの検証
  - 実装したソフトウェアが仕様を満足するかを調べること
- 類似技術：ソフトウェアテスト
  - コンピュータのプログラムを実行して、正しく動作するかを確認する作業
  - ソフトウェア検証は（プログラムを実行することなく）プログラムを何らかの方法で解析して正しく動作するかを確認

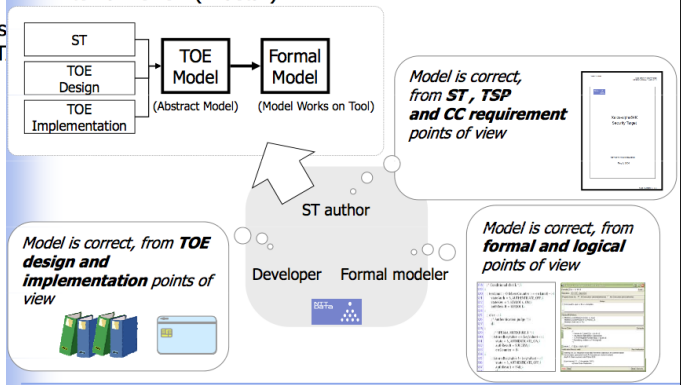
Smartcard security development using formal method tool SPIN

2. Key to succeed

15th of

Naohis
NTTDAT

- Internal Review (in detail)



検証

- **全体検証**
  - **システムと当たられた全体性質（仕様）と一致を判定**

    システム ＝ 全体性質（仕様）

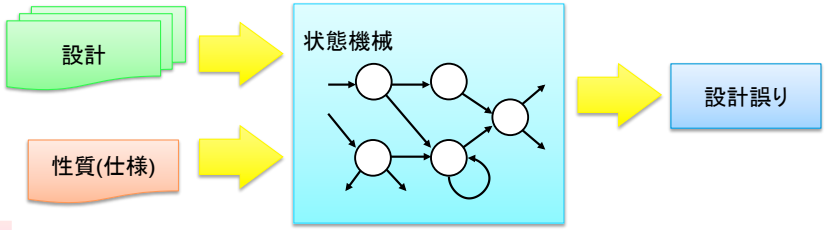
    - **例：プロセス計算の等価関係**
- **部分検証**
  - **システムがある性質を満足することを判定**

    システム ⊃ 部分性質

    - **例：モデル検査（Model Checking）**

モデル検査

- システムやプログラムの動作を表すグラフ(例:状態機械)に対して、調べたい性質(時相論理で記述)の成立するかを判定する

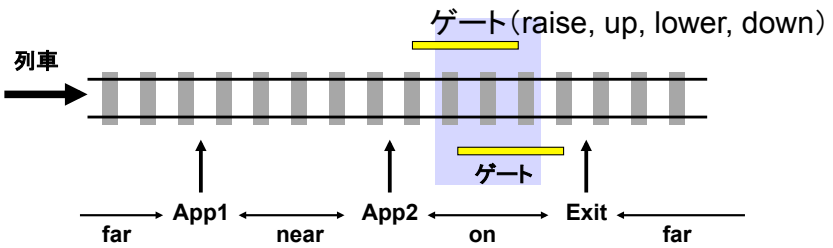


検証

- プログラムが満足すべき性質
  - safety : 初期状態から到達可能な状態*S*で常に条件*P*(*S*)が成り立つ (always *P*)
  - progress : 初期状態から有限回の遷移後に条件*Q*が成り立つ (eventually *Q*)
- モデルチェッキング(モデル検査)
- プログラムの正しさの検査=
  - 状態遷移グラフ(モデル)上で上記が満たされているかどうかの検査

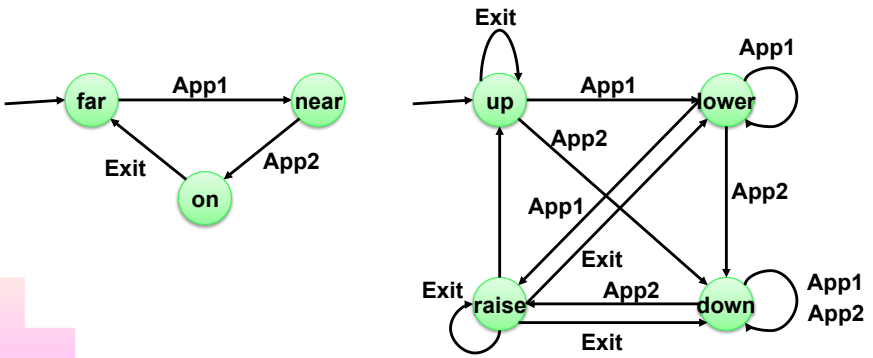
例題:踏切

- 列車が近づくと踏切(ゲート)をおろす



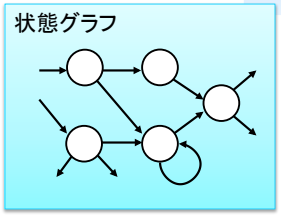
例題:列車の状態遷移

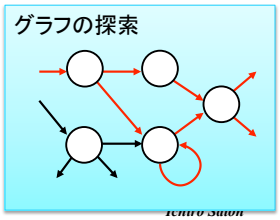
- 列車の状態遷移図



モデルチェッキングの手順

- システム全体の状態遷移グラフを構成
  - 各状態遷移グラフの積(カルテシアン積)
  - 到達不能状態とアークの削除
- 検証手法
  - 調べたい性質を状態遷移に従って状態間を移動
  - 到達した各状態で性質が成立するかを判定





モデルチェッキング

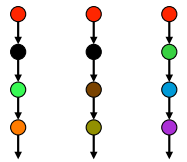
- 2000年前後にソフトウェア検証手法として流行った
  - ツールの普及と、計算機の高性能化が背景
  - オリジナルは1980年代初め に登場(Clarke & Emerson'82など)
  - ハードウェア(論理回路)の検証では日常的に使われている
    - BDD方式
- ソフトウェアに要求されるのは生産性よりも信頼性

モデルチェッキングの特徴

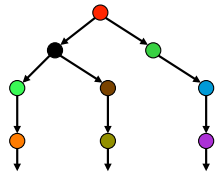
- 有限状態システムを検証
- 状態空間を網羅的に探索
- 部分的な性質を満足するか否かを判定
- 自動処理可能
- 時相論理に対応し、リアクティブ・並列システムを判定可能
- 効率的な判定アルゴリズム
- 検証できなかったときは反例を構成

性質の記述

- 線形時相論理
  - 次の瞬間(ステップ)は高々一つ
  - 一本の無限の時系列



- 分岐時相論理
  - 次の瞬間(ステップ)は一つ以上
  - 無限の時系分岐列



時相論理

- 時相論理とは
  - 状態の遷移や時間の経過の観点よりシステムの性質を記述するための論理体系
- 代表的形式
  - 線形時間時相論理
    - LTL（Linear Time Temporal Logic)
  - 分岐時間時相論理
    - CTL（Computation Tree Logic）
    - プロセス論理
    - μ計算（μ-calculus）

様相論理

- 様相論理
- 必然性と可能性の論理
- 時相論理
- 知識と信念の論理
- 義務論理
- ダイナミック論理
- 内包論理

様相論理

- 米国哲学者Saul A. Kripkeが考案した論理体系

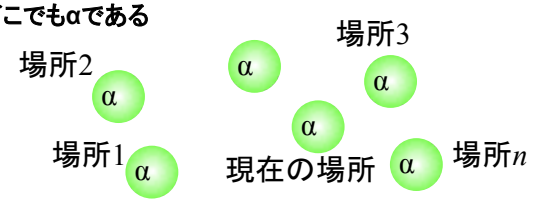
可能性:	～は必然である	～は可能である
時間:	いつでも～である	～であるときがある
場所:	どこでも～である	あるところで～である
確実性:	～でなければならない	～かもしれない
信念:	～を知っている	～を信じている
	$\Box P$	$\Diamond P$

可能世界意味論

- □α　◇α
  - 現在の場所・時刻とは別の場所・時刻におけるαの真理値を考慮する
  - 場所・時刻　…　可能世界（possible world）
  - 可能世界ごとに論理式を解釈

可能世界意味論

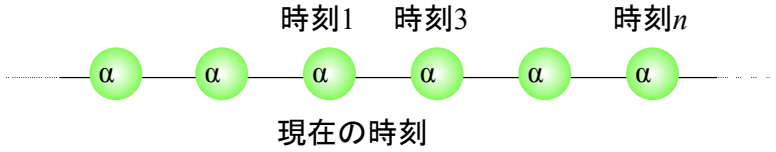
- □α … どこでもαである



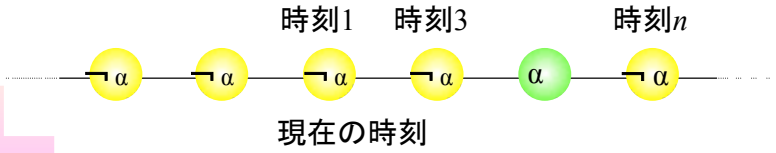
- ◇α … どこかではαである

可能世界意味論

- □α … いつでもαである

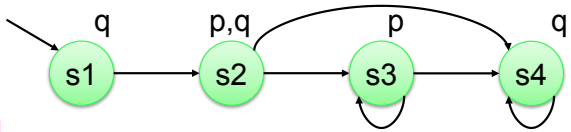


- ◇α … いつかはαである



状態グラフ (クリプキ構造)

- 状態集合: S
  - S = {s1, s2, s3, s4}
- 初期状態集合: I ⊆S
  - I = {s1}
- 遷移関係: R ⊆S×S (注. 必ず次状態が存在するものとする)
  - R = {(s1, s2), (s2, s3), (s2, s4), (s3, s3), (s3, s4), (s4, s4)}
- 原子命題の割当: L: S ↦ 2^A
  - 原子命題の集合 A = {p, q}



Kripke構造

- $K = \langle S, R, L \rangle$

S: 状態の集合

R: 状態間の遷移関係

$R \subseteq S \times S$

L: 状態から原子論理式の集合への写像

L(*s*) は、状態 $s \in S$ において成り立つ原子論理式の集合を与える。

論理式

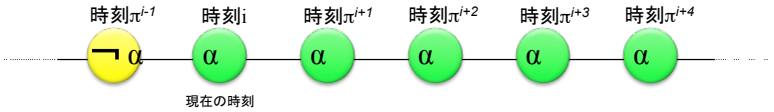
PLTL（命題線形時間時相論理）

ϕ ::= $P \mid \neg P$
| $\phi \wedge \phi$
| $\phi \vee \phi$
| $\bigcirc\phi$ 次の状態でϕが真
| $\Box\phi$ 今後ϕが常に真
| $\Diamond\phi$ 将来のいずれかの時点で真

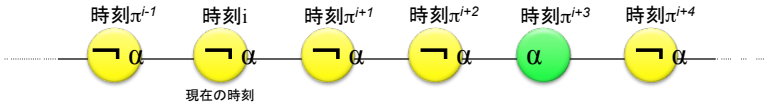
P: 原子論理式

線形時相論理の可能世界意味論

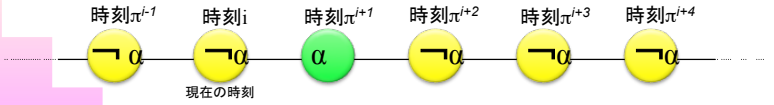
- $\pi^i \models \Box\alpha$ … πの移行は常にαである



- ◇α … いつかはαである



- ○α … 次の時点はαである



意味論

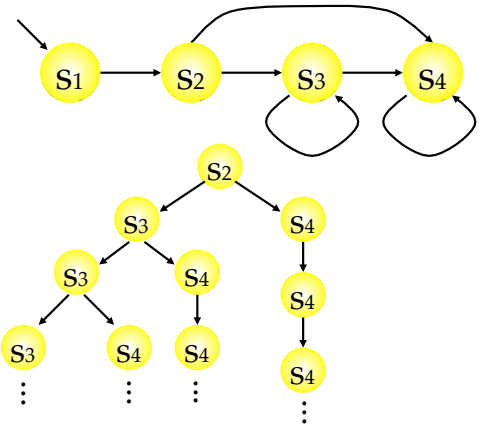
- $\models \neg P$ iff $P \in L(s_0)$
- $\models P$ iff $P \in L(s_0)$
- $\models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
- $\models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$
- $\models \bigcirc\phi$ iff $\pi^1 \models \phi$
- $\models \Box\phi$ iff $\pi^i \models \phi$ for any $i \geq 0$
- $\models \Diamond\phi$ iff $\pi^i \models \phi$ for some $i \geq 0$

- $\models \phi$ --- π は ϕ のモデルである。

$\pi \models \Box\phi$ **iff** $\pi \models \phi \wedge \bigcirc\Box\phi$
$\pi \models \Diamond\phi$ **iff** $\pi \models \phi \vee \bigcirc\Diamond\phi$

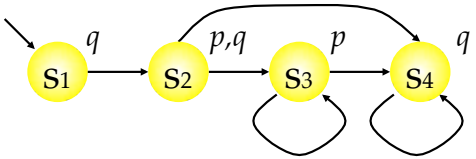
分岐時相論理

- CTL式の真偽値
- 状態グラフ（state graph）上の状態とそこから始まるパスを考える



分岐時相論理の解釈

- CTL式の真偽値
  - CTL式　が状態sで成立する(　がsで真):$s \models \varphi$
  - 原子命題，∨，∧，¬ (not)，→ :その状態だ　けで真偽値が決まる

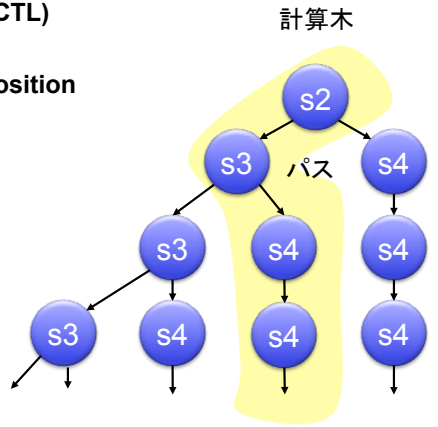


分岐時相論理

Computational Tree Logic (CTL)

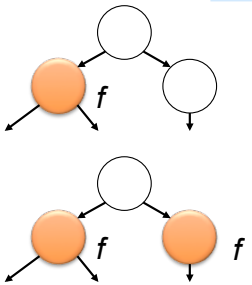
- シンタックス (*f*,*g* はCTL式)
  - 原子命題 atomic proposition
  - ¬*f*, *f* ∧ *g* , *f* ∨ *g* , *f* → *g*
  - EX*f*, EF*f*, EG*f*, E(*f* U*g*)
  - AX*f*, AF*f*, AG*f*, A(*f*U*g*)

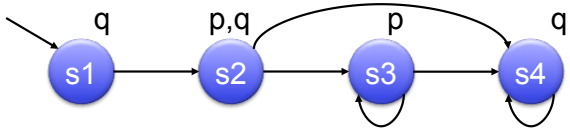
E: あるパスで
A: すべてのパスで
F: いずれ
G: ずっと
U: ～まで



CTL例

- EX*f*
  - fがある次状態で成り立つ
- AX*f*
  - fがすべての次状態で成り立つ



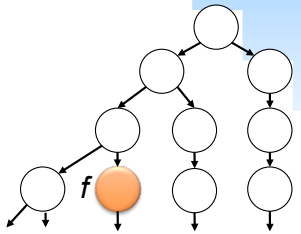


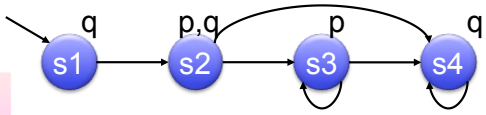
EXp　EXp　EXp
AXp

EXq, AXqは？

CTL例

- EF*f*
  - fがあるパスでいずれ成り立つ
- AF*f*
  - fがすべてのパスでいずれ成り立つ

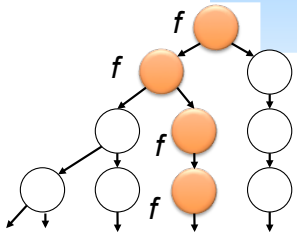




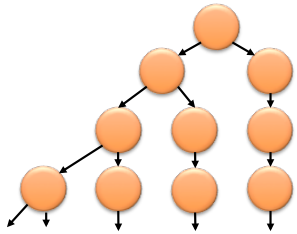
EFp　EFp　EFp
AFp　AFp　AFp

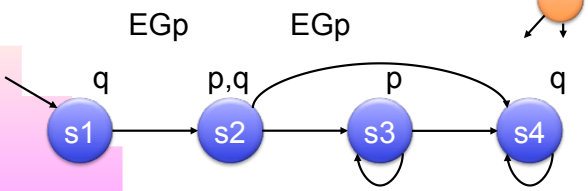
CTL例

- EG*f*
  - fがあるパスで常に成り立つ



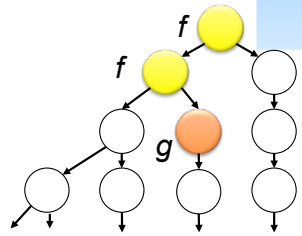
- AG*f*
  - fがすべてのパスで常に成り立つ



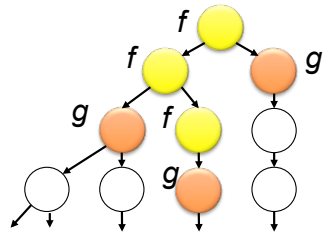


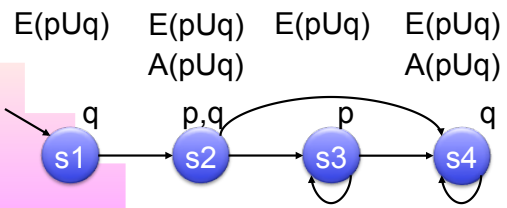
CTL例

- E(*f* U*g*)
  - あるパスについて，どこかでgが成り立ち，それまでfが成り立ち続けている



- A(*f* U*g*)
  - すべてのパスについて，. . .



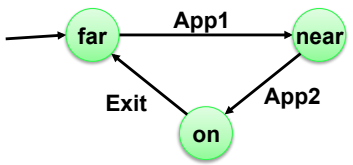


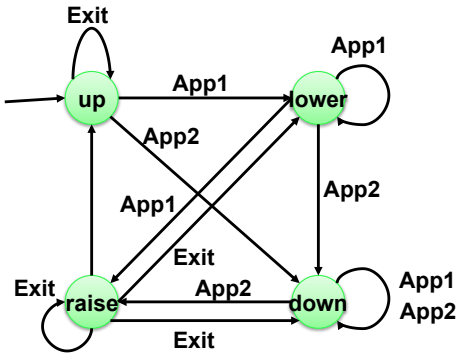
例題：踏切

- 遷移系が満たすべき性質を時相論理 (Temporal Logic)で記述
- Safety: 今からずっと、今からずっと、○○であり続ける であり続ける
  - AG (on AG (on ⇒ down) down)
  - いつでも、電車がゲート上にあれば、ゲートはおりている いつでも、電車がゲート上にあれば、ゲートはおりている
  - AG AX true AG AX true
  - システムがデッドロックしない
- Liveness：将来のある時点で、○○ が成立する
  - AG(down AG(down ⇒ AF up) AF up)
  - ゲートがおりていれば、いつかはゲートがあがる ゲートがおりていれば、いつかはゲートがあがる

例題：列車の状態遷移

- 列車の状態遷移図



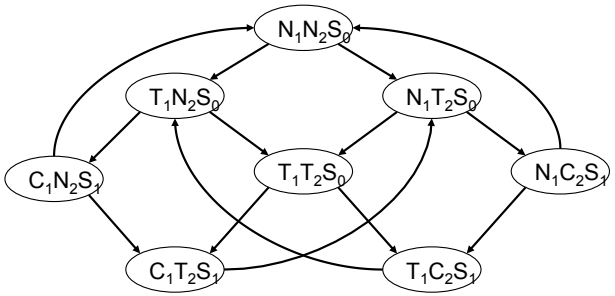


相互排除問題

- 二つのプロセス間で共有変数を排他的にアクセス
- 各プロセスは三つの状態をもつ
  - Non-critical (N)
  - Trying (T)
  - Critical (C)
- 排他制御するためのフラグを導入(S_0またはS_1)
- 初期状態両プロセスともにNon-critical状態(N_1 N_2)、フラグはS_0とする

$$N_1 \rightarrow T_1 \qquad N_2 \rightarrow T_2$$
$$T_1 \wedge S_0 \rightarrow C_1 \wedge S_1 \;\|\; T_2 \wedge S_0 \rightarrow C_2 \wedge S_1$$
$$C_1 \rightarrow N_1 \wedge S_0 \qquad C_2 \rightarrow N_2 \wedge S_0$$

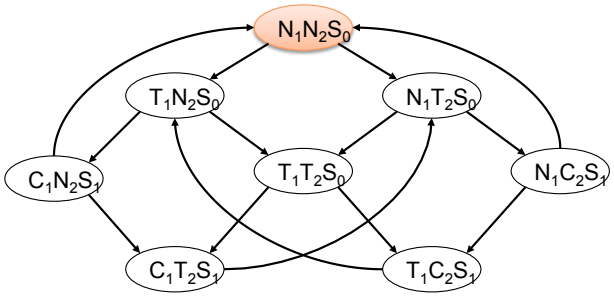
相互排除問題



$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$

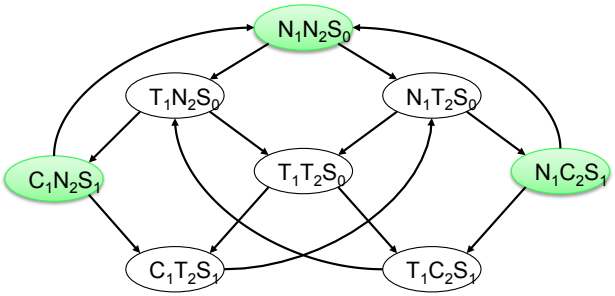
No matter where you are there is always a way to get to the initial state

相互排除問題



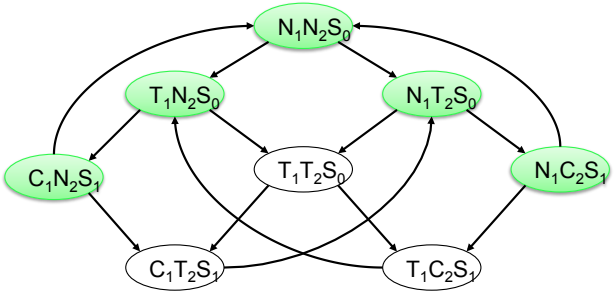
$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$

相互排除問題



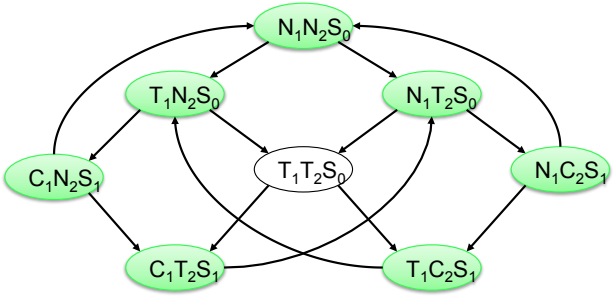
$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$

相互排除問題



$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$

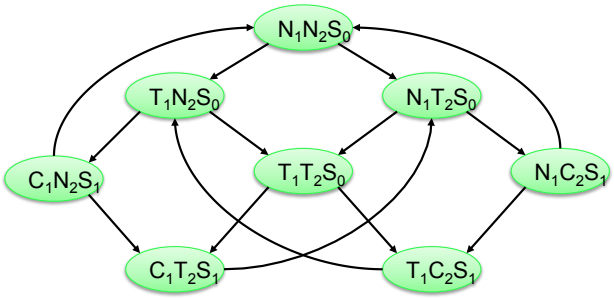
相互排除問題



$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$

Ichiro Satoh

相互排除問題

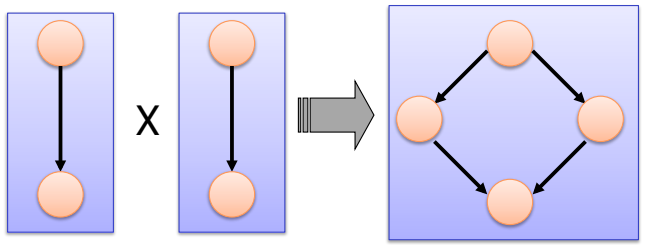


$K \models AG\ EF\ (N_1 \wedge N_2 \wedge S_0)$

Ichiro Satoh

状態爆発問題

- 「モデル規模の増大にともない，探索すべき状態数が急激に増大する」という問題



状態数10の構成要素10個からなるシステム
状態の総数　10^{10} = 100億

Ichiro Satoh

アルゴリズムと計算可能性

- 検証命題を記述する論理体系ごとにアルゴリズムが相違
- CTL論理:
  - 効率的なアルゴリズムが存効率的なアルゴリズムが存在
- LTL論理:
  - 特定の実行系列に関する非常に複雑な命題を記述できるが、式のサイズに関して指数関数的にコストが上がるアルゴリズム

アルゴリズムと計算可能性

- PSPACE-complete
  - 問題のサイズに対して多項式オーダのメモリを使用して解ける問題の中で最も難しい問題のクラスに属するということである
- CTL*
  - 分岐時間と線形時間の操作を組み合わせた論理
  - LTLとアルゴリズム的複雑さが等しい

PromelaとSPIN

- http://spinroot.com/spin/whatispin.html
- Promela : 有限状態プロトコル記述言語
  - メッセージパッシング，共有メモリ
- SPIN : プロトコル検証ツール
  - 擬似実行
  - 完全実行
  - Supertrace実行(大きな問題の一部検証)

Promela記述言語

- 基本概念
  - プロセス
  - 共有メモリ(大域変数)
  - メッセージ送受信(チャネル)
- C的syntax

例題1 (共有メモリ)

```
bool done1 = false;
bool done2 = false;
int x = 0;
proctype incrementer1(){ int t;
  t = x;
  x = t + 1;
  done1 = true; }
proctype incrementer2() { /* ほぼ同様 */ }
init {
  run incrementer1();
  run incrementer2();
  done1 && done2 --> assert(x == 2); }
```


構文

- *E* --> *F*
  - 条件*E*が成り立つのを待って*F*を実行
  - `done1 && done2 --> assert(x == 2);`
  - 実はあらゆる式はその式の「実行可能条件」なる条件を待つ
    - 論理式の「実行可能条件」はその論理式
    - 代入式の「実行可能条件」はtrue
  - 実は --> と ; は全く同じ意味

構文

- proctype A(…) { … }:
  - プロセスの雛形を定義
- run A(…):
  - プロセスAを生成して実行
- mtype = { id_1, id_2, … };
  - メッセージタグの種類を宣言（プログラム中に一度）
  - 本質的には各id_iに一意な定数を割り当てている
- chan *id* = [*N*] of { T_1, T_2, … };
  - メッセージをためるqueue
  - 各要素の型は(T_1, T_2, …)
  - このチャネルは*N*個まで要素を溜めることが出来る
  - FIFO (チャネルに入った順に出てくる)

構文

- *E*!*F* : *E*に*F*を送信
  - 実行可能条件: *E*が満杯でない
- *E*?*tag*(*x*, *y*, …) : *E*の先頭からメッセージ*tag*(?, ?, …)を取り出し，変数*x*, *y*, …を取り出された値にbindする
  - 実行可能条件: *E*の先頭のメッセージが上記パターンに合致する
- 注: 実はパターンの構文はもっと一般的
  - foo(1, *x*, 3) など
  - mtypeは単なる特殊なデータ型．tagのないメッセージも許される

構文

- if
  :: G_1 --> E; E; …; E
  :: G_2 --> E; E; …; E
  …
  fi
- 意味
  - G_iのうちどれかが実行可能になるまで待つ
  - 実行可能なものを非決定的にひとつ選択し，実行
- do
  :: G_1 --> E; E; …; E
  :: G_2 --> E; E; …; E
  …
  od
- if文とほとんど同じ．繰り返すところだけが違う

例題2（メッセージ）

```
mtype = { read, reply };
chan pq = [1] of { mtype };
chan qp = [1] of { mtype, byte };
proctype P() {
  byte v;
  pq!read;
  qp?reply(v)
  --> printf("v = %d\n", v); }
proctype Q() {
  byte x = 20;
  pq?read --> qp!reply(x) }
init { run P(); run Q(); }
```


コマンド

- spin 〈オプション〉〈ファイル名〉
- オプション
  - -s メッセージ送信を表示
  - -r メッセージ受信を表示
  - -p 全ての状態変更を表示
  - -g 全ての大域変数変更を表示

SPIN の情報
http://spinroot.com/spin/whatispin.html

- 全ての可能な状態を並べ上げ，指定した〈性質〉が成り立つことを検証
  - spin –a 〈ファイル名〉
  - pan.cをコンパイル&実行
  - エラーが起きたら，
    spin –t 〈ファイル名〉
    でエラーに至るパスを再現

性質の記述

- assert(*P*)
  - 書かれた場所で*P*が成り立つこと(“always *P*”型)
- progress
- accept
- LTL formula
  - Linear Temporal Logic formula

アルゴリズム

- 状態グラフの構成（循環グラフでもよい）
- 全状態を検査(graph 探索と同値)
- *W* = 初期状態集合;
  C = {}
  while (*W*が空でない)
    pick *s* in *W*;
    *s*に割り当てられたassertを検査
    for all *s'* s.t. *s* -> *s'*
      if *s'* ∉ *C*
        W = *W* + { *s'*}; *C* = *C* + { *s'* };
- 性質を状態遷移の流れに応じて飛ばす（処理の流れを再現）
- すでに到達済みの状態にはマーカーをつけて重複を防ぐ

アルゴリズム

- 1状態のサイズ × 到達可能状態の数
- 少し大きなプロトコルでは容易に，可能メモリ量を突破する(「有限ならよい」というのは近似ですらない)
- どうせメモリが溢れる場合にも“best effort”をする(Supertraceアルゴリズム)
  - 状態遷移グラフ全体を公平にカバーする(unlike 深さ優先)
  - おとずれた状態をなるべく少ないビット数(2 bit!)であらわす

設計

- モデルチェッキングツールが受け付け可能な形式へのデザインの変換
  - 多くの場合、単純にコンパイルする
  - そうでない場合・・・
    - 時間とメモリの限度がある
    - 不用不適切な項目を削除、抽象化が必要

設計

- デザインが満たすべき特性を決定
- 論理形式で与えられることが多い
- 時相理論(temporal logic)の使用
  - システムの動作が時間越しでどう展開するかを記述
- 完全性（completeness）
  - モデルチェッキングは、あるモデルが与えられた設計を満たしているかチェックするための手段であるが、与えられた設計がシステムが満たすべきすべての特性を満たしているかどうかを判断することは不可能である

検証

- 実践では人間の補助が必要
  - 理想では完全な自動化
  - 検証結果の分析
    - 検証が否定的な結果
      - 不正なシステムモデルや不正な設計
      - エラートレースが提供される
    - エラートレースの分析により、システムの修正＆モデルチェキングアルゴリズムへの再適用
  - 検証が失敗して終了
    - モデルサイズがメモリに対し大きすぎる
    - モデルチェッカのパラメータ変更によるモデルの調整（追加的な抽象化）

他の手法

- 記号表現と部分集約で検証できるシステムのサイズは飛躍的にのびた
- しかし、実際のシステムを扱うには不十分→工夫
  - Reasoning
  - Abstraction
  - Symmetry
  - Induction

Reasoning　推論

- 構成推論～単純な形式
  - 構成要素間で独立
  - 局所性を利用して、全体の正当性を推論
    - システムが各局所特性を満たす
    - 局所特性の連結が、全体の設計を含める
- 構成推論～複雑な戦略
  - 構成要素間で依存関係
    - 一つの構成要素の特性を検証する際、他の構成要素について仮定を立てる
    - 他の構成要素が不正だった場合、後でその仮定は破棄

Abstraction 抽象化

- データパスを含むリアクティブシステムの推論
  - データパス
    - 算術演算，論理演算などのデータ処理を中心に行う回路
  - リアクティブシステム
    - 外部環境とインタラクティブに作用し合うことを目的とするシステム(自動販売機など)
- システムの実際のデータ値と抽象化データ値(小さな集合)間のマッピング
  - 抽象化したものはサイズがずっと小さくなる
  - 検証が単純になる

Symmetry　対称性

- 状態爆発問題の削減
  - 縮約モデルは、時相論理式で表されたオリジナルモデルの特性の検証を単純化するために用いられる
- 対称性があるとは・・・
  - 状態遷移図を保存する自明でない置換グループが存在することを意味する
- 置換グループは、システムの状態空間上の等価関係の決定と、状態空間の縮約に用いられる

Induction　帰納

- パラメータで表わされるデザイン
- パラメータのパラメータ値に関するデザインの集合がfamily
- 一群の任意のメンバーの振る舞いを表するinvariant process　の形式を提供
- この不変量を用いることで、一群のメンバーすべての特性を一度に調べることができる
- この帰納的な独立変数は、不変量が適切な表現かを検証するのに使われる