モデル検査における妥当性確認を目的とした並行システムの表現手法

Visualization Method of Concurrent Systems for Validation in Model Checking

竹内亮太郎* 粕谷英人† 大久保弘崇‡ 山本晋一郎§

あらまし システムを意図どおりにモデル化することは，モデル検査を行う前提であるため重要な作業である．しかし，システムのモデル化は人手による部分が大きいため，誤りが生じる可能性がある．本稿はモデルが意図どおりであるかを確認することを目的として，モデルを直観的に理解しやすいグラフで表現し，理解を支援するための表現手法を提案する．具体的には，インタリーブなどの表現を提案する．また，この表現手法に基づく表現を行うツールを作成した．

1 はじめに

システムの複雑化に伴い，その信頼性を保証する検査技術の要求が高まっている．とりわけ，並行システムは複数の逐次プロセスが互いに影響しているため，その信頼性を確認することは困難である．並行システムに対する網羅性の高いテストケースを作成しようとすると膨大な数となり，実用的な時間で検査をすることはできない．

この問題を解決する手段としてモデル検査 [1] が注目されている．モデル検査は，(1) システムを有限状態機械としてモデル化し，(2) 満たすべき性質を記述し，(3) システムが (2) で与えられた性質を満たすか否かを検証する，という流れで行われる．

システムのモデル化や満たすべき性質の記述は人手によって行われるため，誤りが生じる可能性がある．その中でもシステムを意図どおりにモデル化することは，モデル検査を行う前提であるため重要な作業であり，その支援が必要である．本稿ではシステムのモデルが意図どおりであることを「モデルが妥当である」という．

本研究の目的は，モデルが妥当であることの確認を表示によって支援することである．妥当性確認が困難な，システムの並行動作部分の理解を支援するためにモデルの表現手法を提案する．

本稿の構成を以下に示す．2 章はモデル検査について述べ，既存のモデル検査器を紹介する．次に既存のモデル検査器によるモデルの表現の問題点について述べる．3 章は本研究で提案する妥当性確認のための表現手法について説明する．4 章は実装したツールの詳細を説明する．5 章は本研究の評価を行い，最後にまとめと今後の課題について述べる．

2 モデル検査と妥当性確認

本章では既存のモデル検査器を紹介する．また，モデル検査における妥当性確認と，それを行うために必要な表現手法を既存モデル検査器の特徴とともに議論する．

2.1 モデル検査における妥当性確認

モデル検査における妥当性確認とは，モデルが作成者の意図どおりであるかを確認することである．モデルが妥当であるかを確認する方法として，モデルの振る舞

*Ryotaro Takeuchi, 愛知県立大学大学院 情報科学研究科, take@yamamoto.ist.aichi-pu.ac.jp

†Hideto Kasuya, 愛知県立大学 情報科学部, kasuya@ist.aichi-pu.ac.jp

‡Hirotaka Ohkubo, 愛知県立大学 情報科学部, ohkubo@ist.aichi-pu.ac.jp

§Shinichiro Yamamoto, 愛知県立大学 情報科学部, yamamoto@ist.aichi-pu.ac.jp

いをシミュレートすることや，モデルを直観的に理解しやすいグラフとして表示することが有用である．並行システムにおいては，逐次プロセスのモデルをグラフとして表示するだけでは並行動作部分を理解するには不十分である．並行合成することで得られるシステム全体のモデルをグラフとして適切に表示する必要がある．

並行システムの動作理解の上で重要な特徴として，動作が任意の順で現れるインタリーブと，共有資源の利用で発生するクリティカルセクションが挙げられる．

インタリーブ インタリーブとは，並行に動作するプロセスの各処理を任意の順番で実行する技法である．各逐次プロセスの処理がインタリーブすることにより，並行システムの動作が理解しにくくなる．そのため，各逐次プロセスの動作がインタリーブしている箇所を正確に理解する必要がある．

クリティカルセクション クリティカルセクションとは，並行に動作している各プロセスが同時期に共有資源を使用すると破綻をきたす部分である．クリティカルセクションでは，排他制御を行うことでアトミック性を確保する必要がある．複数の共有資源に対する排他制御は複雑なので，不具合が発生する要因になりやすい．そのため，クリティカルセクションはシステムの重点検査箇所である．よって，クリティカルセクションを把握することは重要である．

2.2 既存のモデル検査器

既存のモデル検査器についてそれぞれの特徴を述べる．

2.2.1 Labelled Transition System Analyzer

LTSA (Labelled Transition System Analyzer) [5] は Jeff Magee らによって開発されたモデル検査器である．対象システムは LTSA 専用の言語である FSP (Finite State Process) [6] で記述される．LTSA は FSP で記述したモデルを LTS (Labelled Transition System) として表示する．LTS は状態を表すノードと，ある状態から次の状態への遷移を表すエッジから構成される．エッジには状態遷移が起こるためのアクションがラベルとしてつけられる．LTS は有向グラフと見ることができる．

例 1 (プリンタシステム) 2 人のユーザ Alice (`alice`) と Bob (`bob`) が 1 台のプリンタを共有するシステムをプリンタシステムとする．プリンタの共有は排他的に制御される．ユーザはプリンタを使用する権利を要求し，権利を確保できたら使用して，使用し終わったらプリンタを解放する．対応する FSP を以下に示す．

```
1:  RESOURCE = (acquire->release->RESOURCE).
2:  USER = (printer.acquire->use->printer.release->USER).
3:  ||PRINTER_SHARE = (alice:USER||bob:USER||{alice,bob}::printer:RESOURCE).
```

例 1 の LTSA による表示を図 1 に示す．LTSA はノードを直線上に配置する．上の 2 つのグラフはユーザのモデルを，中心のグラフはプリンタのモデルを，一番下のグラフはシステム全体のモデルを表す．システム全体のグラフは，各逐次プロセスのモデル (以降，各逐次モデルと表記する) を並行合成することにより得られる．

現在の状態を表すノードと直前のアクションを表すエッジを赤くすることにより強調されている．アクションを実行すると状態が遷移する．そのため，アクションの実行を繰り返すことでモデルの状態遷移を視覚的に把握することができる．

また，LTSA にはモデルの振る舞いをシミュレートするアニメータ機能 (LTSA-Animator) が備わっている (図 2)．LTSA-Animator の左側には実行したアクションの履歴が表示される．右側には全てのアクションが表示されており，現在の状態で実行可能なアクションにはチェックがつく．実行可能なアクションを選択すると，状態が遷移し，各逐次モデルのグラフ (以降，各逐次グラフと表記する) の対応するノードとエッジの色が変わる．しかし，並行モデルのグラフ (以降，並行グラフと

表記する) のノードとエッジの色は状態 0 から変わらない.

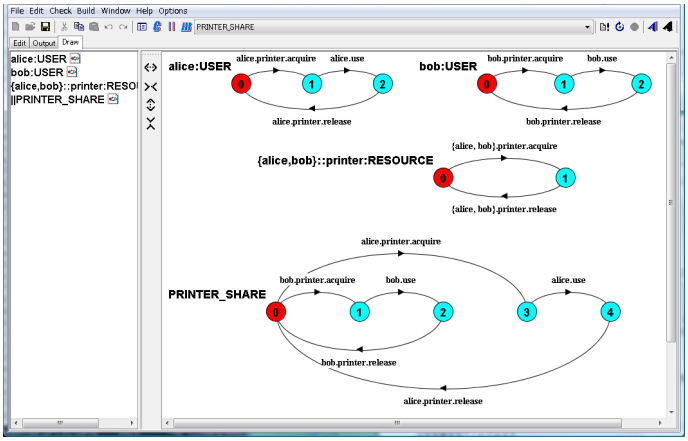


図 1　LTSA

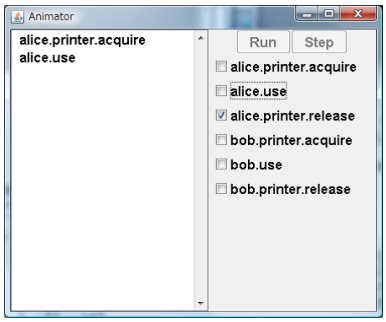


図 2　LTSA-Animator

2.2.2　Spin

Spin [2] は Holzamann らによって開発されたモデル検査器で，検査対象のシステムは Promela(Process Meta Language) 言語で記述される．作成したモデルの振る舞いをシミュレートする機能がある．Xspin は Spin の GUI フロントエンドで，シミュレーションの進行をシーケンス図で表す．例 1 の Xspin による表示を図 3 に示す．左が Promela 言語を記述するウィンドウ，右上がシーケンス図，右下がシミュレーションの実行トレースである．

2.2.3　UPPAAL

UPPAAL [3] はスウェーデンの Uppsala 大学とデンマークの Aalborg 大学によって共同開発されたモデル検査器で，GUI でモデルを作成できる．例 1 の UPPAAL による表示を図 4 に示す．画面右に作成したモデルのグラフが表示されている．ノードは状態を表し，エッジは状態から状態への遷移を表す．並行グラフを表示する機能はない．モデルの振る舞いをシミュレートする機能があり，シミュレーションの進行は右下のシーケンス図で表される．

2.3　既存のモデル検査器における表現の問題点

並行システムの特徴であり理解を妨げる要因として 2.1 節で述べたインタリーブ，クリティカルセクションがあげられる．しかし，既存のモデル検査器によるモデル

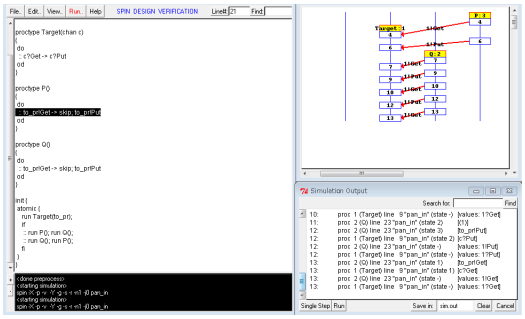


図 3　Xspin

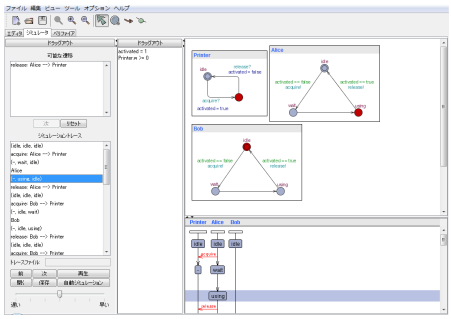


図 4　UPPAAL

の表現では，上記のパターンを適切に表現できていない．例えばLTSAによる表現では，ノードを直線状に並べるため処理の順番を把握しにくく，インタリーブしている遷移が直観的に把握しにくいという問題がある．また，実用的なシステムをモデル化すると，インタリーブによって状態数が爆発的に増加し，理解しにくいモデルとなってしまう．UPPAALやXspinは並行グラフを表示する機能がないため，システム全体の動作を把握しにくい．そのため，インタリーブしている動作やクリティカルセクションを直観的に把握しにくい．2.1節で述べた2つに加えて，以下の2つもシステムの動作理解がむずかしくなる要因である．

プロセスが同じ動作を複数回繰り返す場合
同じ動作を繰り返している箇所を全て表示すると，グラフ全体において繰り返し部分の占める割合が高くなる．その結果，グラフ全体が見にくくなるという問題がある．

プロセス中のサブプロセスがモデルの動作を理解する上で重要である場合
モデルを作成する際に，プロセスの概要を表すダイアグラムを記述して，詳細をサブプロセスとして別のダイアグラムに記述するという方法がとられることがある．そのように作成されたモデルを理解する際には，大まかな動作を理解した上で詳細な動作へ理解を進めるのがよい．しかし，そのための表示を支援する機能は既存のモデル検査器には見つからなかった．

以上のことを解決するために，それぞれの問題に対応した表現手法が必要である．

3 妥当性確認のための表示手法

2.3節で述べたように，既存の検査器によるモデルのグラフ表現ではグラフが直観的にわかりにくく，意図とモデルを対比させて妥当性を確認することがむずかしいという問題点がある．本章では，モデルの妥当性を確認しやすくするための表現手法として，モデルのグラフの特定構造に適した表現手法を提案する．

3.1 モデルのグラフの特定構造に適した表現

モデルのグラフで，そのままの表現では理解が困難な特定の構造について，それぞれの構造に適した表現手法を提案する．以下の4つの表現手法を考える．

3.1.1 インタリーブパターン

並行に動作している遷移をまとめて表示する表現手法をインタリーブパターンとする．インタリーブパターンの適用箇所は，合成したプロセスの全逐次プロセスが同期している遷移から，次の全逐次プロセスが同期している遷移までの並行動作箇所とする．並行動作箇所に同期している遷移はない．また，初期状態から最初の同期遷移が起こるまでは並行動作箇所なので，インタリーブパターンの適用箇所とする．並行動作箇所で遷移が終わる場合もある．インタリーブパターンの適用例として例2をあげる．

例 2 (ProcessABC) 逐次プロセス `ProcessA`，`ProcessB`，`ProcessC` によって構成される並行システムを `ProcessABC` とする．`ProcessA`，`ProcessB`，`ProcessC` は，それぞれ並行して起こるアクションとして `a1`，`b1`，`c1` と `c2` を持つ．また，それぞれ同期して起こるアクションとして `s`,`e` を持つ．`ProcessABC` は，構文的には `s` と `e` で $2 \times 2 \times 2$ の8箇所のインタリーブがあるが，実際に合成した状態空間では図5のように2箇所となる．対応するFSPを以下に示す．... は省略を表す．

```
1: ProcessA = (...->s->a1->e->...->s->a1->e->...).
2: ProcessB = (...->s->b1->e->...->s->b1->e->...).
3: ProcessC = (...->s->c1->c2->e->...->s->c1->c2->e->...).
3: ||ProcessABC = (ProcessA||ProcessB||ProcessC).
```

図 5 の表現は状態が増加し，煩雑なグラフとなりやすい．そこで図 6 のような表現を提案する．この表現はステートチャート図を参考にしている．図 6 の表現では，ProcessA と ProcessB と ProcessC の直積が並行システムの状態を表す．このように表現することにより，煩雑なグラフとなることを防ぐことができる．

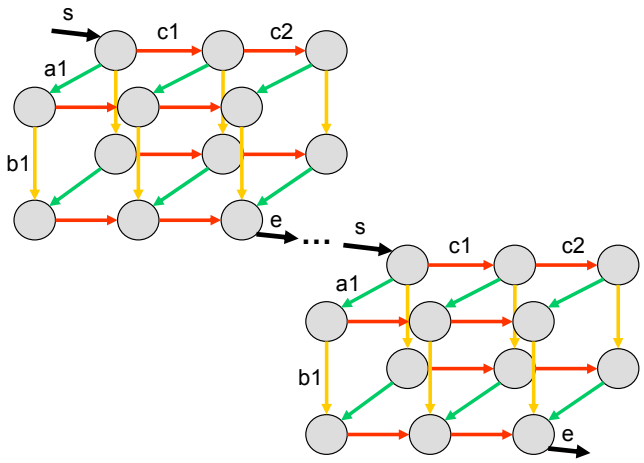


図 5　ProcessABC の表示例

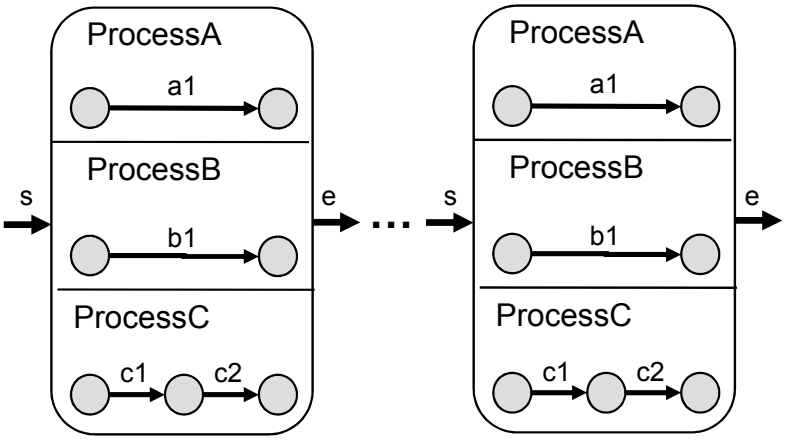


図 6　インタリーブに着目した表示

3.1.2　アノテーションパターン

逐次プロセスのアノテーションを合成プロセスに反映する表現手法をアノテーションパターンとする．アノテーションとして逐次プロセスの強調したい状態を付与して合成プロセスに反映することで，合成プロセスのグラフ表現において直観的に逐次プロセスの強調したい状態を把握することができる．クリティカルセクションなどにおいて有用である．アノテーションパターンの適用例として例 1 のプリンタシステムをあげる．

例 1 を LTSA で表示すると図 1 となり，並行グラフを拡大すると図 7 となる．クリティカルセクションであるプリンタを使用している箇所を把握しづらい表示であ

る．アノテーションとして共有資源であるプリンタを使用している状態を付与し，それを合成プロセスに反映した表示が図 8 である．アノテーションとして付与した状態を枠で囲み明示する．図 8 では共有資源を要求するアクションと開放するアクション，クリティカルセクション部のアクションを把握しやすくなっている．

モデル作成者は元々，図 8 のようにモデルを考えているが，既存のモデル検査器ではクリティカルセクションを考慮した表示をしない．そのため，クリティカルセクションがわかりにくくなる．ここではアノテーションとして共有資源を使用している状態を付与することで，クリティカルセクションを把握しやすくしている．状態 0 は共有リソースがすべて解放されている状態なので囲まれていない．

また，クリティカルセクションが複数ある場合は，クリティカルセクションごとに枠の色を変更することで区別することができる．

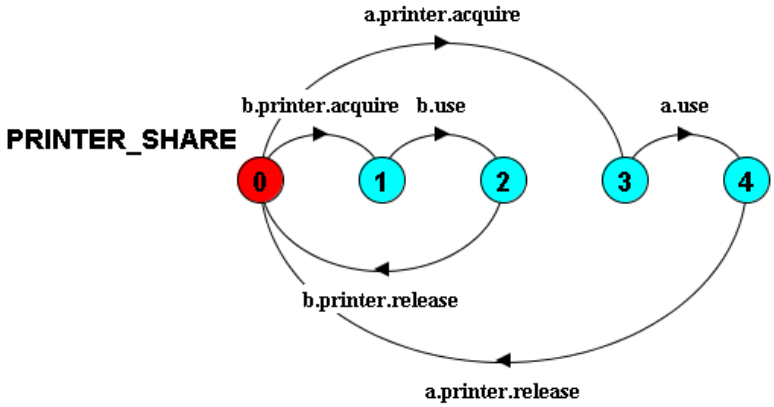


図 7　LTSA による例 1 の並行グラフの表示

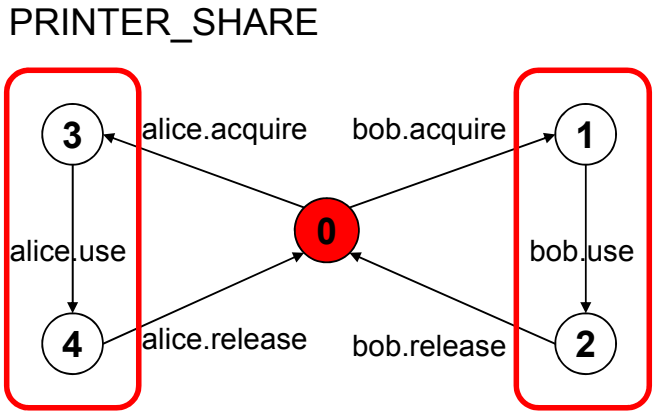


図 8　クリティカルセクションに着目した表示

3.1.3　繰り返しパターン

同じ入出力の遷移を繰り返している箇所に着目する表現手法を繰り返しパターンとする．繰り返しパターンの適用例として例 3 をあげる．

例 3 (カウントシステム)　数をカウントするシステムをカウントシステムとする．

inc 操作で数が増加し，dec 操作で数が減少する．

例 3 を LTSA で表示すると，図 9 のようになる．同じ遷移の繰り返しを表示すると，グラフ全体において繰り返し部分の占める割合が高くなってしまう．そのため，表示スペースが狭くなり，グラフが見にくくなる．そこで図 10 のように，繰り返している部分を畳み込んで表示することにより，繰り返し処理している部分を把握しやすくする．また，畳み込むことにより，繰り返している箇所以外の表示スペースが広くなり，グラフが見やすくなる．

初期状態のノードから順番に見ていき，同じ入出力の遷移をするノードが 2 つ以上連続していたら繰り返し箇所とし，終端ノード，もしくは初期状態になったら終了する．以上のようにグラフの構造から繰り返している部分を判断できる．

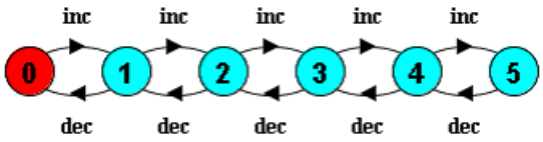


図 9　LTSA によるカウントシステムの表示

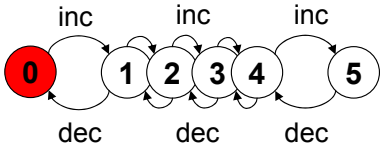


図 10　繰り返しに着目した表示

3.1.4　階層構造パターン

プロセス中のサブプロセスに着目する表現手法を階層構造パターンとする．大規模なモデルを作成する際に，プロセスの概要を表すダイアグラムを記述して，詳細をサブプロセスとして別のダイアグラムに記述することが多い．そのような場合，プロセスの概要と詳細の両方がわかるような表示方法が望ましい．LTSA は FSP の記述にサブプロセスを使用できるが，LTS に変換するとサブプロセスの情報がなくなる．そこで，サブプロセスの情報を表示することで，プロセスの大まかな動作を把握しやすくなると考えた．このパターンの適用例として，文献 [4] の 8 章に登場するクルーズコントロールシステムをあげる．

例 4 (クルーズコントロールシステム)　クルーズコントロールシステムとは，自動車のアクセルペダルを踏み続けることなくセットした速度を維持するシステムである．クルーズコントロールシステムの動作を以下に示す．

- エンジン動作中に on ボタンが押されると，現在の速度を記録し，その速度を維持する
- アクセル，ブレーキ，off ボタンが押されると，速度維持を中止するが，その設定された速度は記録しておく
- resume ボタンが押されると，記録していた速度を維持する

このシステムは 5 つのプロセスでモデル化される．ここでそのうちの 1 つである Cruise controller に着目する．Cruise controller は INACTIVE，ACTIVE，CRUSING，STANDBY のサブプロセスがある．Cruise contoller の振る舞いを以下に示す．

- INACTIVE のときにエンジンをかけると，clearSpeed イベントがおき ACTIVE になる
- ACTIVE のときに on ボタンが押されると現在の速度を記録し，速度維持を開始し CRUSING になる
- CRUISING のときに off ボタン，ブレーキ，アクセルが押されると速度調整を中断し，システムは STANDBY になる
- 任意のサブプロセスでエンジンをオフにすると INACTIVE になる

```
01:set DisableActions = {off,brake,accelerator}
02:CRUISECONTROLLER = INACTIVE,
03:INACTIVE =(engineOn -> clearSpeed -> ACTIVE
04:           |DisableActions -> INACTIVE
05:           ),
06:ACTIVE   =(engineOff -> INACTIVE
07:           |on->recordSpeed->enableControl->CRUISING
08:           |DisableActions -> ACTIVE
09:           ),
10:CRUISING =(engineOff -> disableControl -> INACTIVE
11:           |DisableActions->disableControl->STANDBY
12:           |on->recordSpeed->enableControl->CRUISING
13:           ),
14:STANDBY  =(engineOff -> INACTIVE
15:           |resume -> enableControl -> CRUISING
16:           |on->recordSpeed->enableControl->CRUISING
17:           |DisableActions -> STANDBY
18:           ).
```

例 4 の Cruise contoller を LTSA で表示すると図 11 のようになる．サブプロセスの情報は正当性の検査には必要でないためなくなっている．そのため LTSA による表示ではサブプロセスを把握しにくい．サブプロセスに応じて速度維持が作動するときの判断が変わるので，妥当性を確認する上でサブプロセスを把握することは重要である．サブプロセスを把握しやすくするために，サブプロセスを反映させた表示方法が図 12 である．図 11 の LTSA による表示と比較すると，サブプロセスを把握しやすくなっている．

並行システムに適用する場合は，その並行システムから任意の逐次プロセス 1 個を選び，それを反映する．

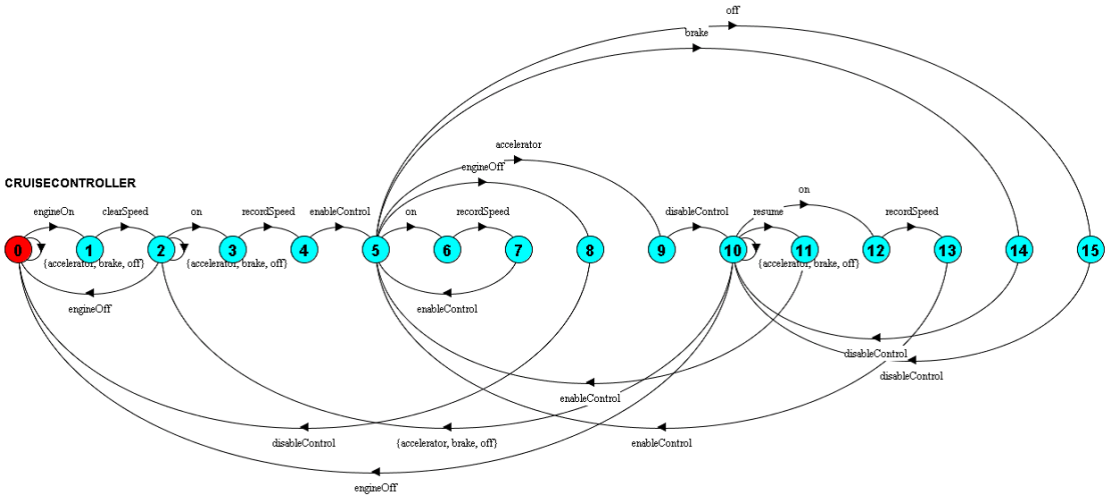


図 11　LTSA による CruiseController の表示

4　実装

本稿で提案する表現手法に基づいた並行システムの直観的な可視化を行うツールを実装した．Java 言語によって実装し，描画ライブラリとして Graphviz [7] の Java

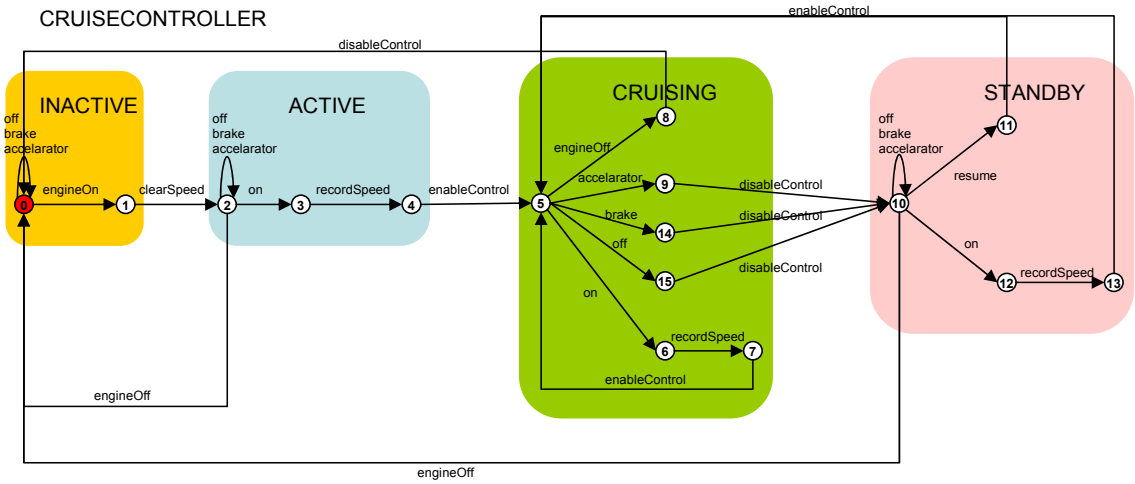


図 12　階層構造に着目した CruiseController の表示

実装である Grappa [8] を利用した．本ツールのシステム構成を図 13 に示す．

本ツールは FSP を入力とし，JavaCC [9] を用いて FSP の解析器を作成した．本ツールの画面構成を図 14 に示す．並行システムのモデルのグラフ表示画面，各逐次プロセスのグラフ表示画面，履歴表示画面の 3 つの画面より構成される．3.1 節に示したモデルのグラフの特定構造に適した表現を行う．アノテーションパターンのみパターン情報を与える．

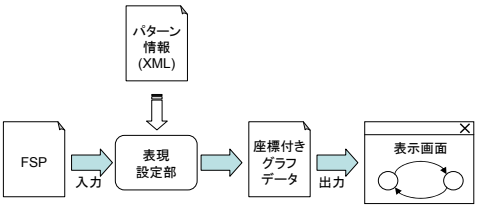


図 13　本ツールのシステム構成

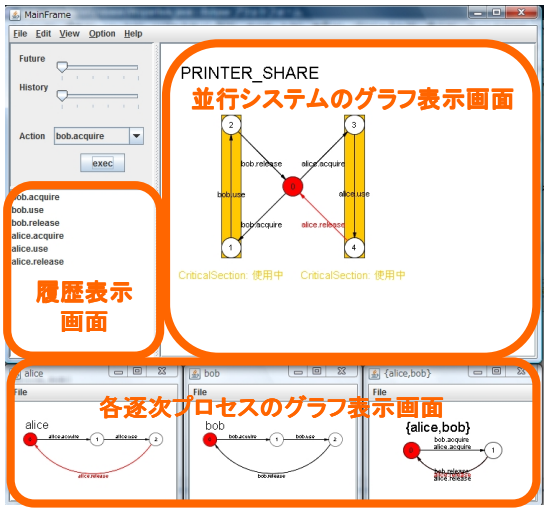
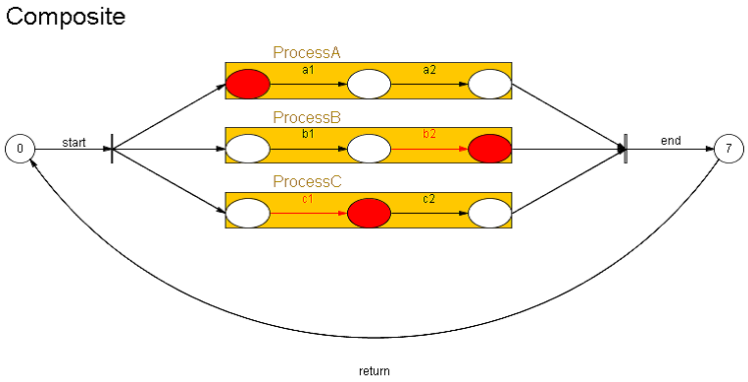


図 14　本ツールの画面構成

4.1　インタリーブパターン

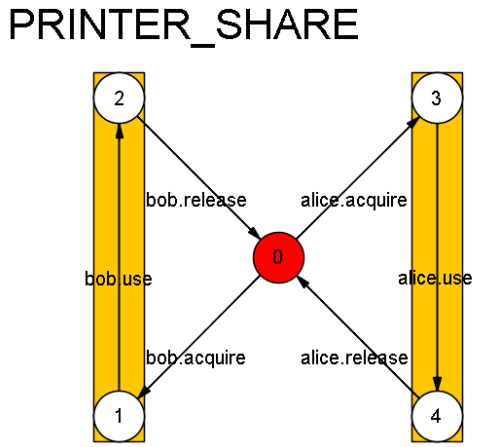
逐次プロセス ProcessA，ProcessB，ProcessC から構成される並行システムを Composite とする．並行して起こるアクションとして a1 と a2，b1 と b2，c1 と c2 があり，同期して起こるアクションとして start，end，return がある．本ツールでインタリーブパターンを適用すると図 15 のように表示される．




```
01:<?xml version="1.0"
encoding="utf-8"?>
02:<graph>
03: <anotation process=
"{alice,bob}::pprinter:RESOURCE">
04:  <node>1</node>
05: </anotation>
06:</graph>
```

図 16　パターン情報の XML 例

図 15　本ツールでインタリーブパターンを適用した表示



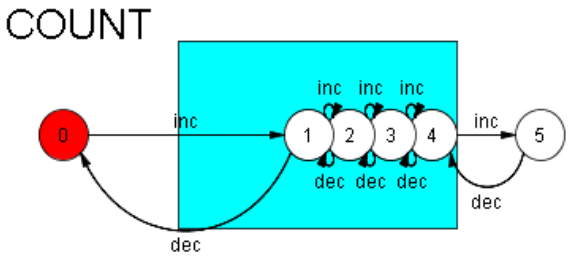


図 17　本ツールでアノテーションパターンを適用した表示

図 18　本ツールで繰り返しパターンを適用した表示

4.2　アノテーションパターン

例 1 に本ツールでアノテーションパターンを適用した表示を図 17 に示す．アノテーションとしてプリンタを使用している状態を付与している．枠で囲まれている箇所がクリティカルセクションであり，クリティカルセクションを直観的に把握しやすくしている．パターン情報として与えている XML を図 16 に示す．また，クリティカルセクションの検出は人手で行っている．

4.3　繰り返しパターン

例 3 に繰り返しパターンを適用した表示を図 18 に示す．同じ動作を繰り返している箇所を畳み込んでいる．

4.4　階層構造パターン

例 4 の `Cruise controller` に階層構造パターンを適用した表示を図 19 に示す．例 11 の LTSA の表示と比較すると，サブプロセスを直観的に把握しやすくなっている．

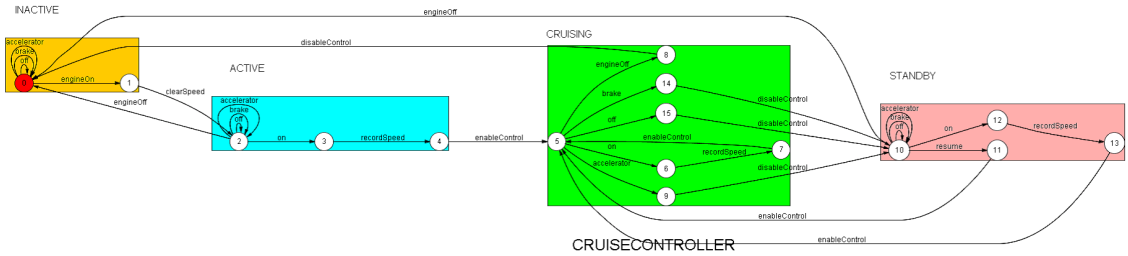


図 19　本ツールで階層構造を適用した表示

5　評価

本ツールと既存のモデル検査器の機能について比較した．モデルの妥当性を検査する機能について比較した．比較する検査器は，2 章で紹介した LTSA，Xspin，UPPAAL とする．

表 1　比較結果

評価項目	本ツール	Xspin	UPPAAL	LTSA
シミュレーションが可能	○	○	○	○
各逐次グラフを表示	○	×	○	○
並行グラフを表示	○	×	×	○
特定構造に適した表示が可能	○	×	×	×

1. 作成したモデルに対して振る舞いのシミュレーションをする機能: 全てのツールが作成したモデルをシミュレーションする機能を持っている．
2. 並行システムを構成する逐次プロセスのモデルをグラフとして表示する機能: Xspin は表示する機能がない．UPPAAL は各逐次グラフを表示する．本ツールと LTSA は各逐次グラフ，並行グラフの両方を表示する．
3. 並行システムのモデルをグラフとして表示する機能: 本ツールと LTSA は表示する機能がある．
4. 3.1 節で述べた表示を行う機能: 本ツールのみが行うことができる．

本ツールは 3 章で提案したモデルのグラフの特定構造に適した表示を行うことができる．また，モデルの振る舞いをシミュレートすることが可能である．現在の状態と直前のアクションを表すエッジを赤くすることで強調し，モデルの状態遷移を視覚的に把握しやすくしている．並行グラフも色が変わるようにしており，システム全体の動作も把握しやすい．以上の点において，LTSA と UPPAAL によって出力されるモデルのグラフより理解しやすく，妥当性の確認を行いやすいと考える．

文献 [4] の 3 章から 10 章までに出てくる例題 61 個への適用結果を表 2 に示す．2 章は逐次プロセスの例しかないため省略とした．

表 2 より 38 箇所にパターンが適用できた．しかし一つのグラフに対して複数のパターンが適用できるため，適用数は多くはない．これは，非常に簡単な例題が多いことなどが理由に挙げられる．また，排他制御に関する例題が比較的多かったため，アノテーションパターンの適用数が一番多くなった．

6　おわりに

本研究では，モデル検査において妥当性を確認しやすくするための表現手法を議論した．そして，妥当性を確認しやすくためにモデルのグラフの特定構造に適した表現方法を提案し，可視化を行うツールを実装した．具体的にはインタリーブやク

表 2　適用結果

章	適用数			
	インタリーブ	アノテーション	繰り返し	階層構造
3 章	2	2	0	1
4 章	0	1	2	2
5 章	0	1	5	0
6 章	0	5	0	0
7 章	0	7	0	1
8 章	0	0	0	2
9 章	1	0	0	0
10 章	0	0	5	1
計	3	16	12	7

リティカルセクションなどの特定構造に着目し，それぞれの構造に適した表現を選択することにより，モデルのグラフを直観的に理解しやすくできることを示した．その結果，モデルの妥当性を確認しやすくなったと考える．

本稿では 4 つの構造に特化した表現を提案したが，より多くの構造に着目することで，よりモデルの妥当性を確認しやすくなると考えられる．また，インタリーブパターンの適用箇所は，すべての逐次プロセスが同期してから再び全ての逐次プロセスが同期するまでとしているが，一部のプロセスが同期するような場合も扱えるようにすることで，より多くのシステムを理解しやすくなる．以上を今後の課題とする．

参考文献

[1] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled, “Model Checking”, MIT Press, 1999.
[2] Gerard J. Holzmann, “The Model Checker SPIN”, IEEE Trans. Soft. Engin., Vol.23, No.5, pp.279–295, 1997.
[3] Johan Bengtsson, Fredrik Larsson, “UPPAAL – a Tool Suite for Automatic Verification of Real-Time Systems”, DoCS Technical Report Nr 96/67, Uppsala University, ISSN 0283-0574, January 1996.
[4] Jeff Magee, Jeff Kramer, “Concurrency: State Models & Java Programming”, Wiley, 2008.
[5] LTSA - Labelled Transition System Analyzer, *http://www.doc.ic.ac.uk/ltsa/*.
[6] FSP notation, *http://www.doc.ic.ac.uk/∼jnm/LTSdocumention/FSP-notation.html*.
[7] Graphviz, *http://www.graphviz.org/*.
[8] Grappa, *http://www.research.att.com/∼john/Grappa/*.
[9] JavaCC (JavaCompilerCompiler) Java Parser Generator, *https://javacc.dev.java.net/*.