コンピュータと教育 67－3
（2002．12．13）

Model Checkingを用いた並行プログラミング学習支援システムの試作

鶴見 誠悟† 角川 裕次†

並行プログラムは複数のプロセスが相互作用しながら実行されるプログラムである．プロセスのスケジューリングは実行ごとに異なる場合があるため，非常に多くの実行例が存在する．この理由によりプログラムは各スケジューリングに対して正しく実行されなければならない．しかし無数に存在するプロセススケジューリングを全て把握することは困難であり，正しい並行プログラミングを行うことは難しい．本研究では初心者を対象とした並行プログラミング学習支援システムを試作した．このシステムでは，学習者にプログラムの空欄補充問題を提供する．学習者が解答したプログラムを形式的検証手法の一つであるモデル検証により検証し，全てのプロセススケジューリングにおいてプログラムの正誤判定を行う．またプログラムの直感的な理解を促すために，プロセスの状態や実行の履歴を学習者に表示する．

Implementation of a learning system for concurrent programming with model checking

SEIGO TSURUMI† and HIROTSUGU KAKUGAWA†

A concurrent program is a program such that two or more processes interact each other. Because there are huge number of instances of process schedulings, programs must be correct for each of them. Unfortunately, it is very difficult to imagine all process schedulings, writing a correct concurrent program is difficult. In this paper, we propose a learning system for introductory concurrent programming. In our system, fill-in-blank problems are offered for learners to complete concurrent programs. An answer is checked by model checking system, which is one of formal verification methods, if it is correct or not for any process shcedulings. For intuitive understanding, process states and execution history are displayed on windows.

1 序論

並行プログラムは，正しいコードを書くことの難しさ，適切なデバッグを施すことの難しさなどより，初心者にとって習得が困難なものとなっている．

そのため並行プログラミングのための学習支援環境が必要になってくる．従来のプログラミング学習支援ではプログラムの動作をアニメーションで可視化する [6]，プログラムのコードの読解を支援する [11] といったことは多数行われてきたが，プログラムがエラーなく正しく動作することを理解させる手法はあまり用いられていない．

本研究では並行プログラミング学習支援システムを試作した．本システムでは並行プログラムに形式的検証手法のモデル検証を用いて検証を行い，プログラムのエラー検知を行う．また，プログラムの動作・状態を可視化することで，学習者にプログラムを直感的に理解させることを促す．

† 広島大学大学院工学研究科
Graduate School of Engineering, Hiroshima University

1.1 並行プログラムについて

並行プログラムとは，複数のプロセスが相互に作用しながら次々に切り替わり実行され全体としての機能を実現するプログラムのことである．

並行プログラムは，GUIアプリケーションなどの応答性がよくなる，マルチCPU環境では並列実行によって処理速度が上がるなどのメリットにより，需要が大きく様々な分野で利用されている．例えばウェブブラウザではhtmlファイルをダウンロードしながら同時にそのhtmlの画面を表示することができる．

しかし，以下のような理由により初心者にとって並行プログラミングの習得は困難なものとなっている．

- プログラムの動作・状態の把握が難しい

  並行プログラムは複数のプロセスから構成され，プロセスがお互いに作用しながらプログラムとしての機能を実現する．そのため，プログラム全体の動作・状態の把握が困難である．

- 非常に多くの実行パスが存在し，実行の再現性が低い

  ここで実行パスとは，実行されるプログラム内での実際に実行した命令の順序のことである．並行プログラムでは実行する度にプロセスの切り替わりの順序が異なる場合があり，その場合は実行ごとに異なった実行パスとなる．そのため，同じ実行をもう一度再現しようとしても再現しない場合も多い．

  そして同じ実行の再現が難しいため，エラーの発見，特定が困難になる．例えば，ある実行パスでエラーが発生したとする．そのエラーが他の多くの実行パスで発生するのならばデバッグは比較的容易である．しかしその実行パスでのみエラーが発生する場合，実行パスの再現が困難な場合はエラー原因の特定は難しい．

また実際に並行プログラムを記述する際には次のことに気を付ける必要がある [13, 2].

- 干渉: 干渉とは複数のプロセスが同じ資源に同時にアクセスすることである．これによりプログラムの一貫性を損なってしまう．そこで，あるプロセスがある資源にアクセスしている間は，別のプロセスからはアクセスできないようにする必要があり，これを排他制御という．排他的に処理しなければならないコードの領域のことをクリティカルセクションという．

- デッドロック: 干渉を防ぐためにロックを使って排他制御を行う際，ロックをかける手順によっては全てのプロセスが待ち状態になり，プログラムが先に進まなくなってしまうことがある．この状態をデッドロックという．

- 安全性と健全性: 並行プログラミングでは，逐次プログラミングの場合には考慮する必要のなかったことを考慮する必要がある．一般的に並行プログラミングでは，安全性，健全性というプログラムの性質を確保しなければならない．安全性とは「好ましくない状態にならない」という性質のことである．例えば干渉が起きない，デッドロックが発生しないことを示している．健全性とは「やろうとしていることはいつか必ず行われる」という性質のことである．例えばクリティカル・セクションに入ろうとしたプロセスはいつかは必ず入ることを示している．

このような理由で，正しく動作する並行プログラムを書くことや，適切なデバッグを施すことが難しく，並行プログラムの初心者にとっては習得することが困難となる．

そのため，並行プログラミングに対する学習支援が有効になってくる．

1.2 関連研究

プログラミング学習支援に関する研究はこれまでにも多数行われてきた．

文献 [6] では，並列プログラムの可視化を行っている．アルゴリズムを 2 次元，3 次元アニメーションとして視覚化する枠組みを提供している．文献 [11] ではプログラムのコードとそのコードに対する解説を利用し，ドキュメントを生成するシステムを構築している．このドキュメントは，コードにおけるアウトラインから詳細部へのトップダウン的な構造になっており，読んで理解するのに適切な構造になっている．また文献 [8] では並行プログラミング学習支援ツールを作成している．このツールではプログラムの動作・状態を理解するための手法として，Interleave Matrix とアニメーションを用いている．このようにプログラムの動作を可視化する，プログラムのコードの読解を支援するといったことはプログラムの直感的な理解のためには有効な手法である．

また文献 [9] ではプログラムの正しさの理解を目的とした教材作成システムを構築している．このシステムではホーア論理による表明を用いて逐次プログラムが正しく動作していることを示し，学習者に正しさの理解を促している．しかしこのシステムでは学習用プログラムの判定結果が真偽値のみで表されそれを学習者に提供するので，学習者は得る情報が少なく学習効果が低い．

ここでプログラムがエラーなく正しく動作することを示すには，形式的検証手法を用いてプログラムを検証することが有効である．近年，デバッグやテストを自動的に行うための検証技法を用いたプログラムの検証が盛んに行われている．文献 [1] では，形式的検証ツールである spin[5] を用いて，Java 言語で記述されたプログラムの検証とデバッグを行っている．文献 [7] では，これも同じく Java 言語で記述されたプログラムをカラーペトリネットを用いて，テストと検証を行っている．また文献 [10] では，アスペクト指向プログラムがデッドロックなどの予期しない動作を引き起こすかどうかをモデル検証により検査している．

このようにプログラムの形式的検証においてはモデル検証，ペトリネットなどが用いられている．しかし，ペトリネットはトランジションの発火などといった独特のメカニズムによりプログラムを検証する．そのためプログラムの仕様そのものの表現には適していない．それに対してモデル検証ではプログラムを状態遷移図，プログラムの仕様を論理式で表現しているので，状態遷移図，論理式を理解していればモデル検証を用いることは比較的容易である．そのため本研究ではモデル検証を用いた形式的検証を行っている．

1.3 本研究について

本研究では初心者向けの学習支援を行う．そのためプログラムの動作および状態の可視化を行いプログラムの直感的な理解を支援する．また，形式的検証手法のモデル検証を用いてプログラムのエラー検知を行い，プログラムが正しいか正しくないかの判定を行う．

本稿の構成は以下の通りである．まず第 2 章では形式的検証手法の一つであるモデル検証について簡単に述べる．第 3 章では本研究で試作したシステムについての説明を行う．第 4 章ではシステムの実装について述べ，第 5 章では本研究のまとめと今後の課題について述べる．

2 モデル検証

モデル検証とは形式的検証手法の一つで，プログラムの仕様を論理式で記述し，プログラムがその仕様を満たしているかを判定することにより検証を行う [3, 4]．ここで用いられる論理式は，従来の命題論理に時間の概念を表現する時相演算子を付加した時相論理の一種である CTL(Computation Tree Logic) と呼ばれるものである．CTL 式の意味を定義するセマンティックモデルとして，有限状態機械のモデルと考えられる Kripke 構造と呼ばれる有向グラフを用いる．

2.1 Kripke 構造と Computation Tree Logic (CTL)

Kripke 構造は四つ組 $K = (V, R, I, V_0)$ で定義される．ここで，V は状態の有限集合，$R \subseteq V \times V$ は状態間の遷移関係を表す．ここでの状態間の遷移とは時間の流れを表す．$I : V \to 2^{AP}$ は各状態で真になる原子命題の集合を与える．ただし，AP は原子命題の集合である．また，$V_0 \subseteq V$ は初期状態の集合である．

CTL の論理演算子は，通常の命題論理の論理和 $(+)$，論理積 $(\cdot)$，論理否定 $(\neg)$，含意 $(\to)$，等価 $(\equiv)$ などの命題演算子と時相演算子からなる．時相演算子は，Kripke 構造のある状態からの

- 「全ての遷移系列 (ここで遷移系列とは遷移する状態を並べた列のことである) に対して」という意味を表す**全称記号**(A)
- 「ある遷移系列に対して」という意味を表す**存在記号**(E)

を用い，これらを時間に関する性質を表現する演算子，

- 「次の時刻 (next)」を意味する X
- 「将来いつか (future)」を意味する F
- 「将来常に (global)」を意味する G
- 「ある性質が成り立つまでは別の性質が成立し続ける (until)」を意味する U

などと組み合わせたものである．それらの直感的な意味は次の通りである．

- $EX\phi$: 現在の状態のある遷移先の状態で ϕ が成立する
- $AX\phi$: 現在の状態の全ての遷移先の状態で ϕ が成立する
- $EG\phi$: 現在の状態からのある遷移系列において，常に ϕ が成立する
- $AG\phi$: 現在の状態からの全ての遷移系列において，常に ϕ が成立する
- $EF\phi$: 現在の状態からのある遷移系列において，いつか ϕ が成立する
- $AF\phi$: 現在の状態からの全ての遷移系列において，いつか ϕ が成立する
- $E[\psi U\phi]$: 現在からのある遷移系列において，いつか ϕ が成立し，かつ最初に ϕ が成立するまでは ψ が成立し続ける
- $A[\psi U\phi]$: 現在からの全ての遷移系列において，いつか ϕ が成立し，かつ最初に ϕ が成立するまでは ψ が成立し続ける

プログラムに対してモデル検証を行う場合，プログラムを Kripke 構造で表現し，プログラムの仕様を CTL 式で記述する．そしてプログラムが仕様を満たしているかどうかを Kripke 構造と CTL 式で検証する．

プログラムを Kripke 構造で表現する場合，Kripke 構造の各状態において以下のような値を保持する．

- プログラム内の各変数の値
- 各プロセスで現在実行されようとしている命令を示すプログラムカウンタの値

そして1つのプロセスのプログラムカウンタの値が変化する，つまり1つのプロセスの命令が1文実行されると状態遷移が起こる．このようにしてプログラムの実行に伴い状態が変化していく．

またプログラムの仕様をCTL式を用いて表現する場合は，変数の値，プログラムカウンタの値などを原子命題として仕様記述を行うことになる．

2.2 モデル検証の例

ここで，例として相互排除の問題を考えてみる．このプログラムは，$turn$ という変数に対して2つのプロセスが代入を繰り返すというプログラムである．そのプログラムを以下に記す．ただし各命令の前の値はプログラムのどこを実行しているかを示すラベルとする．**NC** はノンクリティカルセクション，**TC** はクリティカルセクションに入ろうとしている部分，**CR** はクリティカルセクションを表す．

```
    Program Mutual_Exclusion {
        turn = 1;

        Process A {
NCa:        while True do
TCa:            wait (turn = 0);
CRa:            turn = 1;
            end while;
        }

        Process B {
NCb:        while True do
TCb:            wait (turn = 1);
CRb:            turn = 0;
            end while;
        }
    }
```

このプログラムをKripke構造で表現すると図1のようになる．各状態内の値はプログラムで使用される変数の値，各プロセスの現在実行されようとしている命令を示すプログラムカウンタの値である．

これに仕様として以下の2つを与える．

- 安全性 (Safety): $\phi_1 = AG\ \neg(\mathbf{CR_a} \wedge \mathbf{CR_b})$
  任意の状態からの全ての遷移系列において，同時に，プロセスAのラベルが $\mathbf{CR_a}$ になり，プロセスBのラベルが $\mathbf{CR_b}$ になることはない．つまり，プロセスA,Bが同時にクリティカルセクションに入ることはない．

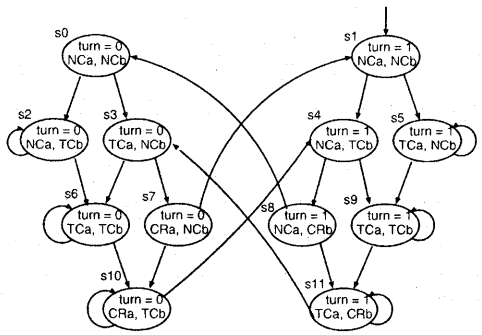

図1: 相互排除のプログラムのKripke構造での表現

- 健全性 (Liveness): $\phi_2 = AG(\mathbf{TC_a} \rightarrow AF\ \mathbf{CR_a})$
  任意の状態からの全ての遷移系列において，プロセスAのラベルが $\mathbf{TC_a}$ ならば，いつかは $\mathbf{CR_a}$ になる．つまり，プロセスAがクリティカルセクションに入ろうとしているならば，いつかは入ることができる．

そしてこの ϕ_1, ϕ_2 の各仕様について検証を行う．まず ϕ_1 であるが，全ての状態において同時に $\mathbf{CR_a}$，$\mathbf{CR_b}$ となっていなければよい．そこで各状態のプログラムカウンタの値を調べてみると全ての状態が条件を満たしている．よって仕様 ϕ_1 は満たされている．次に ϕ_2 であるが，状態 s_6 を見てみるとループが発生している．このループが永遠に続く限り，いつまでたってもプロセスAはラベル $\mathbf{TC_a}$ の命令を実行して，ラベル $\mathbf{CR_a}$ の命令に到達することができない．よってこのプログラムは仕様 ϕ_2，つまり健全性を満たしていないことになる．

3 本システムについて

本システムでは，学習支援の対象として，逐次プログラムをある程度書けるようになった人，並行プログラミングの初心者の人とする．また本システムが学習支援対象とする並行プログラムは共有変数型のプログラムである．

前述したように，並行プログラムは正しいコードを書くことの難しさ，適切なデバッグを施すことの難しさなどの理由で並行プログラミングの初心者にとっては習得が困難である．そのため学習支援というものが有効になってくる．

ここで並行プログラミング学習において何が重要かを考えると以下のようになる．

- プログラムのコードに対してプロセスが現在どういう状態にあり，どういう動作を行っているかを把握する

- エラーが起こらず正しく動作することを確認する
- ただプログラムを眺めるだけでなく実際にコードを記述する

そこで本システムではプロセスが行う変数の参照，代入，1動作ごとの変数の値の変化などを可視化し学習者に提供した．また形式的検証手法の一つであるモデル検証を用いて，プログラムが正しく動作することを検証し，その情報を学習者に提供する．そして学習者に学習用プログラムの空欄を補充させる．こうして学習者はプログラムの動作・状態を把握しつつ，プログラムが正しく動作することを確認し，また実際にコードを記述することでプログラムに対する理解を深めていく．

本研究で試作するシステムは，プログラム言語としてJavaを用いた．これはJavaのネットワークとの親和性を考慮してである．システムはWWW上に構築され，学習者が時間的，空間的制約を受けることなく学習を行えるようにした．学習者はブラウザを通して学習を行う．システムの全体像は図2のようになる．

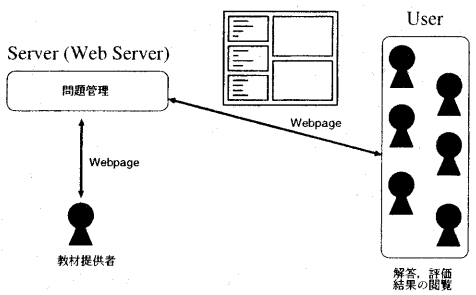

図 2: システムの全体図

サーバでは学習用プログラムの管理を行う．学習者へ表示する部分にはhtmlのページとJavaアプレットを用いた．htmlのページで空欄補充問題を提供し，アプレットでプログラムの検証，動作・状態の可視化を提供する．現時点ではサーバ側の実装はまだ不十分なことと，本稿ではどのように学習を行うかが焦点なので，サーバ部分の説明は省略する．

また本システムにおける学習の流れとしては次のようになる．

1. 学習者にプログラムの空欄を補充させる
2. そのプログラムをモデル検証で検証する
3. プログラムがあらかじめ与えられた仕様を満たしていない場合 (エラーがある場合)，エラーの起こる実行パスを取得する
4. エラーがある場合，エラーの起こる実行パスを実行し，学習者にエラーが起こることを示す
5. 学習者自身によってプログラムを1ステップずつ自由に実行させていき，プログラムがどういう動作をし，どのような状態になるかを理解させる

3.1 学習支援

本システムではプログラムの動作・状態の可視化，モデル検証によるエラー検知を学習支援に用いる．

プログラムの動作・状態の可視化のための手段として以下のものを用いる．

- プロセスが行う変数の参照，代入の表示を行うシーケンス図
- 変数の値表示を行う変数テーブル

並行プログラムにおいて，各プロセスがどういう相互作用をし，その結果変数の値がどのように変化したかを把握することは重要なことである．プログラム全体の動作をアニメーションで表現する (例えば生産者・消費者問題のプログラムを生産者と消費者が動いているアニメーションで表す) 手法も直感的な理解のためには有効である．しかしこれではプログラムの概観が掴めるだけであり，プログラム内のプロセスが相互作用しながら動作することの理解にはつながらない．また個々のプログラムのアニメーションを作成することになると，教材作成の手間が増えることになる．そのため本システムでは自動的に生成可能なシーケンス図および変数テーブルを用いることにした．

シーケンス図は，UML[12]のシーケンス図を参考に作成した (図3)．UMLのシーケンス図はシステム，プログラムなどの動的な振舞を表現するのに有効である．

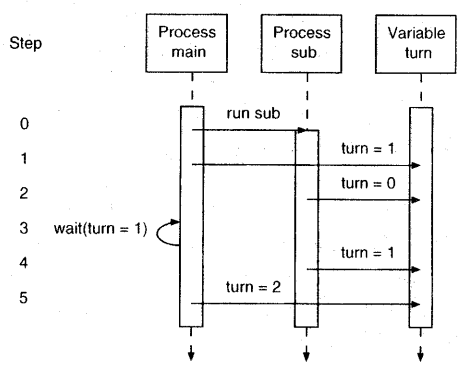

図 3: シーケンス図

また変数テーブルにおいて，変数の値をプログラムの1動作ごとに取得し，表示することにより，現在プログ

ラムがどのような状態になっているかを把握することができる (図 4).

Step	Variable1	Variable2	Variable3
0	0	1	2
1	1	0	2
2	0	0	2
3	1	0	2
4	0	1	4
5	1	2	2
⋮	⋮	⋮	⋮

図 4: 変数テーブル

そしてプログラムが正しく動作するかどうかの検証のためにモデル検証を利用する．プログラムに与える仕様としては例として以下のようなものを用いる．

- 干渉が起こり得ない
- デッドロックが起こり得ない
- 各プロセスの安全性，健全性が保たれている

実際に学習者が見る画面は図 5 のようになる．

図の左側のパネル群で各プロセスのソースコードを表示する．また画面中央上の部分がシーケンス図，下の部分が変数テーブルである．また画面右はエラー表示用のウィンドウである．

ソースコード表示部分では，Play ボタンを押すことで，各プロセスが 1 行ずつ実行されていく．丸のついている行はこれから実行しようとしている行を示す．

またシーケンス図と変数テーブルは，Play ボタンを押していくことで表示が変化していく．

そしてエラー表示ウィンドウでは，モデル検証によりプログラムにエラーが発見された場合，そのエラーの内容を表示する．

そして学習者が間違ったプログラム (仕様を満たさないプログラム) を書くと，モデル検証によりエラーが発生する実行パスを見つけだす．そしてエラーが発生する実行パスを自動的に再実行することで，学習者にエラーの原因を把握させる．

3.2 学習の例

ここで実際の学習の例を示す．現時点で，以下のようなプログラムをサンプルとして作成している．

- 相互排除のプログラムの空欄補充問題
- 共有資源アクセスのプログラム

相互排除のプログラムの空欄補充問題は，相互排除を実現するための仕組みをプログラムで記述している．相互排除を実現している部分のコードを空欄補充することで相互排除の仕組みを理解させる．また共有資源アクセスのプログラムは 2 つのプロセスが共有資源 (この場合は共有の変数) にアクセスするプログラムで，スケジューリングによってはデッドロックが発生するという意図的にバグを仕込んだプログラムである．このプログラムを実行し，デッドロックを発生させることでどのような仕組みでデッドロックが発生するのかを理解する．

ここでは，例として相互排除のプログラムの空欄補充問題を解いてみる．このプログラムはモデル検証の節での例に用いたものと同じである．プログラムに与える仕様としてモデル検証の節の例での安全性の仕様 $\phi_1 = AG\ \neg(\mathbf{CR_a} \wedge \mathbf{CR_b})$ を与える．プログラムの空欄部分はプロセス B のラベル $\mathbf{TC_b}$ の部分である．

まず，図 6 のページで，プログラムの空欄を補充する．ここで例えば間違った答え $wait(turn = 1)$ を入力したものとする．

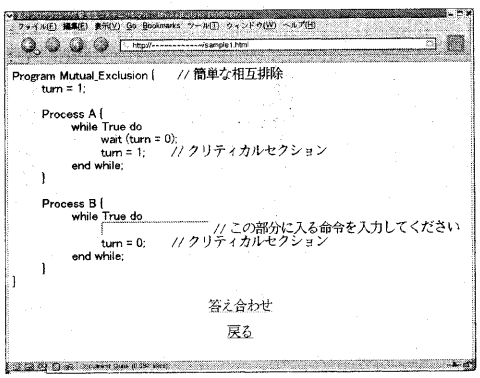

図 6: プログラムの空欄補充の画面

空欄を補充した後，答え合わせのボタンを押すと，システム側でプログラムに対してモデル検証を行い，画面は図 5 に移る．

ここでは，空欄に $wait(turn = 1)$ と入力したが，正しいプログラムでは $wait(turn = 0)$ となっている．そのため，正しく相互排除を行うことができず，安全性という仕様が満たされないことになる．実際プロセス A,B が同時にクリティカルセクション (この場合 $turn$ に値を代入する行) に入ることが可能になっている．

ここで，前述したように並行プログラムは多数の実行パスが存在するため，エラーが発生することなく実行が終了することもあればエラーが発生してしまう実行パス

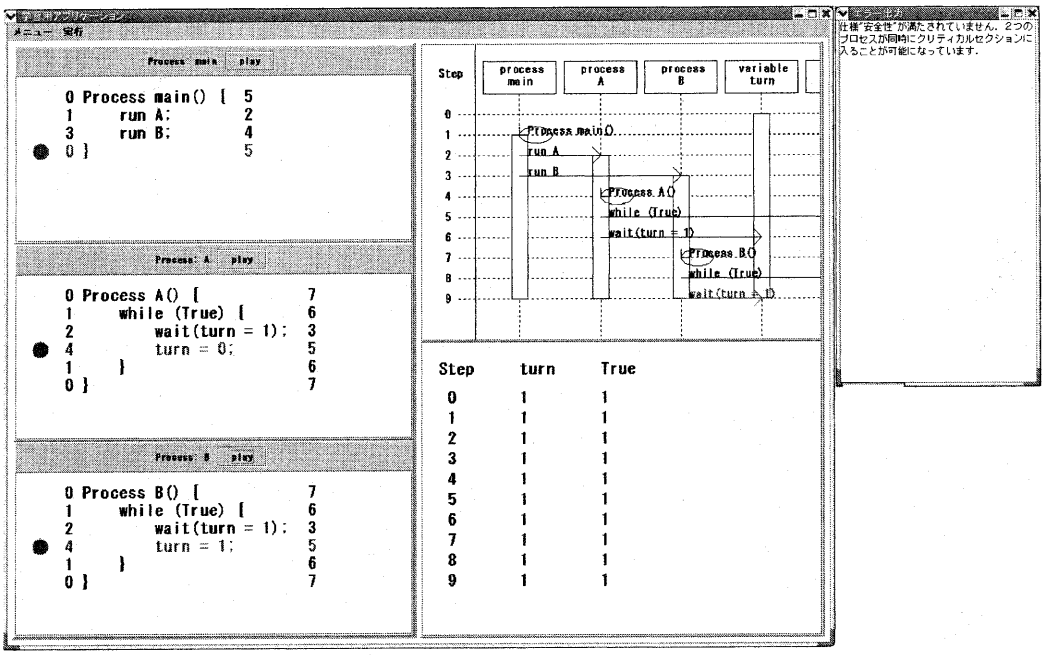

図 5: 学習者の見る画面

もある．そのため学習者がプログラムを 1 ステップずつ自由に実行させるだけでは，エラーの特定が困難である．しかしモデル検証を用いて検証を行っているため，エラーの発生する実行パス (ここではプロセス A,B が同時にクリティカルセクションに入ることができる実行パス) を発見して，学習者に提示することができる．

このようにして学習を進めていくことにより，どのようにして相互排除を実現するのか，エラーがあればどのようにしてエラーが発生したのかを理解することができる．

4 実装

本システムでは，Java のアプレットを用いて学習画面の実装を行い，学習用プログラムの記述フォーマットとして XML を採用した．プログラムの XML ファイルの形式はおおまかに次のようになっている．

```
<Program>
    <Variable>変数</Variable>
    <Specification>仕様</Specification>
    <Body>
        <Process>
            <Statement>命令</Statement>
                .
                .
                .
        </Process>
            .
            .
            .
    </Body>
</Program>
```

タグ Variable 部分でプログラム内で使用する変数を定義し，タグ Specification 部分でプログラムの仕様を記述する．タグ Body 部分では各プロセスのコードを記述する．

学習用プログラムの XML ファイルはあらかじめ教材提供者が用意しておく．ウェブブラウザで学習者が空欄補充したプログラムの断片は文法解析され，学習用プログラムの XML ファイルに組み込まれる (図 7)．

そして，XML ファイルより Kripke 構造を生成する．Kripke 構造はアプレットの中で動的に生成される．そしてこの Kripke 構造に対してモデル検証を行う．モデル検証部分は自作し，学習用アプレットに組み込んでいる．検証の結果，仕様が満たされない場合，エラーメッセージを出力し，エラーの起こる実行パスを取得する．

また学習用画面において，Play ボタンを押してプログラムを実行する際には，押されたボタンのプロセスのプログラムカウンタの示したタグ Statement の部分を読み

込み，そこに書いてある命令を実行する．同時にシーケンス図，変数テーブルに実行した命令，その結果変化した変数などの情報を送り，表示を変化させる．

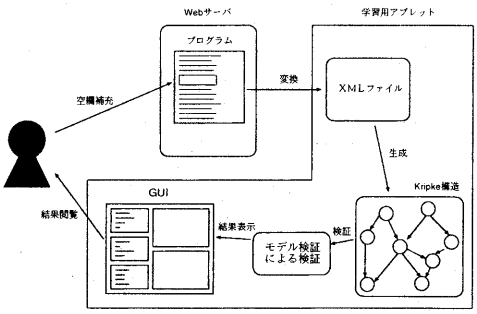

図 7: 実装

5 結論

本稿では，並行プログラミングに対する学習支援システムの試作を行った．モデル検証によりプログラムが正しく動作するかを検証し，その情報を学習者に提供した．またプログラムの動作・状態を可視化し，プログラムがどういう動作を行っていて，どういう状態にあるかを提供した．そして学習者に実際にプログラムの空欄を補充させることで，プログラムに対する理解を促すことができる．このような学習支援により，効果的な学習が行えるようにした．

現在は，学習支援部分 (モデル検証によるプログラムの検証，プログラムの動作・状態の可視化，プログラムの空欄補充) はほぼ完成した．今後行うこととしては，教材生成部分の作成，サンプルの作成などが挙げられる．また図 5 の画面において学習者が自由にプログラムを実行させていく，エラーが発生する実行パスの自動実行を行うといったこと以外にも学習方法を増やし，より効果的な学習が行えるようにする必要がある．

また，システムの評価については実装が完了次第行う予定である．

参考文献

[1] Klaus Havelund, Thomas Pressburger, "Model Checking Java Programs Using Java PathFinder", International Journal on Software Tools for Technology Transfer, Vol.2, No.4, pp.366-381, 2000.

[2] Gregory R. Andrews, "Concurrent Programming Principles and Practice", The BenjaminCummings Publishing Company, Inc, 1991.

[3] Edmund M, Clarke Jr, "Model Checking", The MIT Press, 2000.

[4] Michael Huth, Mark Ryan, "Logic in Computer Science", Cambridge University Press, 2000.

[5] Spin, *http://spinroot.com/spin/whatispin.html*

[6] POLKA, *http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html*

[7] 渡辺 晴美, 徳岡 宏樹, Wu Wenxin, 佐伯 元司, "カラーペトリネットによるオブジェクト指向ソフトウェアのテストと解析方法", 電子情報通信学会, Vol. J82-D-1, No.3, pp. 478-495, 1999.

[8] 星野 拓郎, "並行プログラミング学習支援ツール", 広島大学工学部卒業論文, 2001.

[9] 森 正, "論理的なプログラム理解のためのオンライン教育システム", 広島大学大学院工学研究科修士論文, 2001.

[10] 鵜林 尚靖, 玉井 哲雄, "アスペクト指向プログラミングへのモデル検査手法の適用", 情報処理学会論文誌, Vol.43, No.6, pp.1598-1609, 2002.

[11] Donald E. Knuth, 文芸的プログラミング, アスキー出版, 1994.

[12] 浅海 智晴, "やさしい UML 入門", Pearson Education Japan, 2001.

[13] 戸松 豊和, "Java プログラムデザイン", SOFT BANK Publishing, 2002.