Vol.2017-SE-195 No.13
2017/3/12

モデル検査によるゲームシナリオの不具合発見ツールの提案

木間塚達[†1] 野田夏子[†1]

概要：ゲーム開発において，シナリオの不具合がないことを発見することは重要である．状態空間の網羅的な検査が可能なモデル検査を用いてゲームシナリオの不具合を発見する研究は過去にもなされているが，シナリオ作成者が実際に適用するには困難さがあった．本研究では，シナリオ作成者が実際に利用できるツールとして，ゲームシナリオとテンプレートに従った簡単な性質記述を入力することにより SPIN を利用して不具合を発見するツールを提案する．

A Verification Tool of Game Scenarios using Model Checking Techniques

ITARU KIMATSUKA[†1] NATSUKO NODA[†1]

1. はじめに

ゲーム業界はここ数十年で大幅な成長を遂げてきた．年々ゲームの開発技術が進歩し，ゲーム 1 つあたりの開発規模が大きくなっている．開発規模が大きくなった結果，不具合の数も多くなっており，不具合を発見するための労力が増えている．特にゲームの台本となるシナリオが原因で起こる不具合の見逃しは，そのゲームに重大な影響を及ぼすこともある．したがって，ゲーム開発においてシナリオの不具合の発見は非常に重要であり，特に開発初期において発見することが望まれる．

このゲームシナリオの不具合発見について，モデル検査[1]を利用した検証方法が研究されている．清木[3]は，モデル検査技術のアルゴリズムを利用して，ゲームシナリオの不具合を検証し，有用であることを示した．モデル検査技術は，検証するモデルが仕様を満たしているかを検証する技術で，有限の状態空間を網羅的に探索することが可能である．しかしこの研究は，ゲームシナリオを状態遷移として表現し，モデル検査を用いて検証する方法を理論的に示したものであり，具体的なゲームシナリオの検証にそのまま利用することは困難である．

そこで，本研究では，モデル検査ツール SPIN[2]を用いて，ゲームシナリオをそのまま検査できるツールを提案する．

2. ゲームシナリオのモデル検査

ゲームシナリオは，ゲームの進行を示す台本であり，シナリオ記述言語によって記述される．シナリオ記述言語（以下，言語ないしはその言語で書かれたプログラムをスクリプトと呼ぶ）に必要な機能は，以下である．

- 画面表示の更新
- 内部状態の保持
- 内部状態の更新
- 状態を参照しての条件分岐
- ユーザ入力の内部状態への反映

従来シナリオ記述言語して用いられてきたものは，これら最低限の機能のみを持ったテキストベースの言語であることが多い．図 1 に従来のスクリプトの例を示す．

```
IF flag(31) = 1 THEN GOTO label1502
IF flag(32) = 1 THEN GOTO label1503
GOTO label1504

*label1502
: 31番フラグによって実行されるイベント
SHOWSG "background.01.gif"
SHOWING "character.01.01.gif"
PRINT "メッセージ出力1"
WAIT
PRINT "メッセージ出力2"
WAIT
:
GOTO label1504
|
*label1503
:32番のフラグによって実行されるイベント
SHOWBG "background.02.gif"
SHOWING "character.01.02.gif"
PRINT "メッセージ出力3"
WAIT
flag(33) = SELECT "選択肢1","選択肢2","選択肢3"
:
*label1504
:一般の処理
:
```

図 1 スクリプトファイルの例

†1 芝浦工業大学
Shibaura Institute of Technology

ⓒ 2017 Information Processing Society of Japan

図 1 のようなスクリプトを記述していれば，いわゆるスパゲッティプログラムが生まれる可能性もある．さらに，番号のみで管理された変数を使い，数千から数万のフラットなコードブロック群を条件分岐命令で気ままに移り渡るプログラムを作成していると，変数値の集合で表されるシナリオの内部状態を製作者が追いきれなくなってしまうこともある．その結果，不具合が入り込みやすく，その発見も難しくなる．

このような問題に対して，モデル検査技術を利用してゲームシナリオの不具合を発見する研究がなされている[3]．この研究では RPG やアドベンチャーゲームなどと分類されるインタラクティブなストーリー展開を特徴とするコンピュータゲームのシナリオを対象として，デバッグ時の困難を低減するために，モデル検査の一種である記号モデル検査理論を用いたゲームシナリオの自動検証を試みている．ゲームシナリオを状態遷移モデルとして解釈して取り扱うことによってモデル検査のアルゴリズムを適用し，ある種の典型的なバグを自動的に発見できることを示している．典型的なバグの例を以下に挙げる．

- 実行不能ブロックの存在
- 「ハマり」状態の存在
- ストーリー依存関係の矛盾
- 各種 invariant の矛盾

特に，典型的かつ致命的なバグである「ハマり」状態を検出できることは，ゲームシナリオという範囲においては非常に重要になる．清木は，モデル検査理論をゲームシナリオに適用可能かを調べるために，実際に市販されているゲームのシナリオに対して，シナリオスクリプトを基にグラフを作成，簡約化などの手順を踏み，自作したモデル検査プログラムを利用して検証した．この方法で行った実験で「ハマり」などの不具合が発見できたことを報告している．

しかし，この研究に基づいてゲームシナリオにモデル検査を適用する場合，上記のような手順を踏んで検証しなければならないため，モデル検査の知識と高度なプログラミング能力が必要になってしまい，実適用には困難があると言える．

3. モデル検査

本研究で利用するモデル検査及び SPIN について，本章で簡単に説明する．

3.1 モデル検査とは

モデル検査とは，有限の状態空間を網羅的に，ハードウェアやソフトウェアの設計から導出されたモデルが形式仕様を満たすかを検証する手法である．モデル検査は有限の状態空間で表現できるシステムを網羅的に探索することが可能であるため，システム開発者の意図しない動作を発見するのに非常に効果的である．

有限状態遷移系（有限オートマトン）S と時相論理式 p を入力すると，S が p を満たすか満たさないかを検証する効率の良いアルゴリズムが開発されている．モデル検査はこのアルゴリズムを利用して，モデルが論理式を満たしているかをコンピュータで自動判定する手法である．全数探索を行うので，検査式が成り立っていない場合に，どの状態で不成立になるのか（反例）がはっきりわかる．

3.2 SPIN とは

SPIN[2]は，G.J.Holzmann により提案されたモデル検査のためのツールの 1 つである．その開発は，1980 年代初期から始まるが，現在でもツールの改良がされている．SPIN では並行プロセス及び，個々のプロセスの非決定的な振る舞いを Promela と呼ばれる言語で記述する．そして，性質を論理式で指定することで，記述した振る舞いの中で，その性質が成立するかどうかを自動的に検査する．

時相論理には様々な種類がある．代表的なものとしては，時間の進み方が枝分かれして木構造となるものに対して性質を記述する CTL（Computation Tree Logic, 計算機論理）や，時間の進み方が一直線のものに対して性質を記述する LTL（Linear Temporal Logic, 線形時相論理）がある．他にも色々な種類の時相論理があり，その種類により，記述能力やモデル検査アルゴリズムが異なるのである．SPIN では LTL の論理式として記述した性質を取り扱うことができる．

3.3 Promela とは

Promela（PROcess MEta LAnguage）はモデル検査ツール SPIN のモデル記述言語である．基本的に SPIN は並行システムの論理的一貫性を検証するツールであるため，モデリング言語 Promela では，並行プロセスの動的生成を記述することができる．チャネルを介したメッセージ通信は同期通信，非同期通信の両方ができる．

Promela はプロセス単位でモデルを作成する．active をプロセスの宣言の前に付けることで自発的に動作するプロセスとして，宣言できる．Promela にはラベルと goto 文を使うことができる．if 文と goto 文を利用して，作成した Promela を図 2 に示す．

```
byte cnt=0;
active proctype rand(){
again:
        if
        ::0 < cnt ->cnt--
        ::cnt< 5 -> cnt++
        fi;
        goto again
}
```

図 2　Promela の例

Promela には，java などのように文字列を扱う変数がない

ⓒ 2017 Information Processing Society of Japan

ため，文字列を扱いたい場合は，列挙型の mtype を利用する必要がある．

このような通常のプログラムとは構成が異なるため，一般的開発者が Promela でモデルを記述することは困難である．

3.4 LTL 式とは

LTL 式では，命題論理式に時相演算子（Temporal Operators）と呼ばれる記号をつけて修飾する．

表 1 は時相演算子を表すものである．また表 2 に SPIN における LTL で利用する演算子を示す．

表 1　時相演算子

時相演算子	SPIN の表記	意味
◇	〈〉	いつかは（sometime）
□	[]	常に（always）
X		次の時間（next time）
U	U	まで（until）

表 2　SPIN における LTL で利用する演算子

演算子	文字列	意味
⇒	->	ならば
∧	&&	かつ
∨	||	または
¬	!	否定

基本的な論理式に時間の概念が加わるため，一般開発者が LTL 式を用いて満たすべき性質を記述することは困難である．

4. ゲームシナリオ検査ツール

4.1 SPIN を用いたゲームシナリオの検証

2 章で述べたように，ゲームシナリオの不具合を設計時に検証するためには，モデル検査の利用が有用であると期待される．しかし，先行研究[3]で提案されたような理論を理解した上でシナリオをモデル化し，提案された方法で検証することは困難である．そこで，本研究では，モデル検査エンジンとしては SPIN を利用し，ゲームシナリオを検証することを考える．既存のモデル検査ツールを利用する場合でも，時相論理やモデル検査ツールに入力するための専用の記述言語についての知識が必要である．SPIN を利用する場合であれば，LTL 式や Promela についての知識がなければ利用できない．しかし，シナリオ製作者やゲーム製作者の多くはモデル検査技術に関する知識を持たないと考えられる．そこで，モデル検査エンジンとしては SPIN を利用しつつ，利用者はモデル検査の知識がなくてもモデル検査を行うことができるツールを提案する．

4.2 対象

本研究では，ゲームシナリオをモデル検査することを目的とするため，ゲームジャンルの中でシナリオ以外の要素が最も少ないと考えられる，ノベルゲームを対象とする．またゲームの開発環境はノベルゲームを開発するのに適しているフリーソフトの吉里吉里 Z[4]とし，シナリオを記述するスクリプト言語は吉里吉里 Z で利用されている KAG3[5]とする．

4.3 提案ツールの概要

提案するツールは，SPIN を利用する．しかし，ゲームシナリオとそれが満たすべき性質を，Promela と LTL 式で記述することはゲームシナリオ作成者にとっては困難である．そこで，検査対象として KAG3 で記述されたスクリプトファイルをそのまま入力し，また満たすべき性質をより簡素に表現できる方法を検討する．

満たすべき性質を記述する方法として，テンプレートを定義することでツールユーザが記述しやすい形にする方法を提案する．

ツールの概要は，以下である．

- 入力は，KAG3 で記述されたスクリプトファイルとシナリオが満たしていて欲しい性質をテンプレートに従って記述したテキストの 2 つとする．
- ツール内部では，入力されたスクリプトファイルを SPIN 専用のモデル記述言語 Promela へ，テンプレートに従って記述されたテキストを LTL 式に変換する．
- 出力は，SPIN による検証結果で，シナリオに問題があればその反例を表示し，シナリオに問題がな

ⓒ 2017 Information Processing Society of Japan

ければ問題なしと表示する．

図 3 に提案する不具合発見ツールの概要を示す．

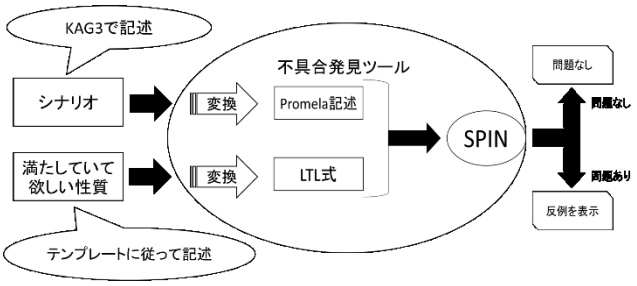


図 3　不具合発見ツールの概要

本研究で提案するツールは吉里吉里 Z で作成したノベルゲームを対象としているが，吉里吉里 Z でのゲーム製作には様々な方法があり，その全ての方法に対応するツールを提案することは困難であるため，あらかじめシナリオ製作上のルールを決めることとする．1 つ目のルールとして，エンディングの状態が判別できるように，ラベル名の先頭の文字列を end で始める．このルールを決めた理由は，詳しくは 4.5.1 で説明するが，SPIN の機能を利用してエンディングの状態をチェックするために必要であるからである．

また，意図しないラベルの移動を防ぐために，1 つ下のラベルへ移行する時以外はラベルの最後の文に停止を意味する [s] タグを原則として付けることにする．

4.4 シナリオの Promela への変換

4.4.1 吉里吉里 Z の KAG3 の構造

本研究で利用する KAG3 で記述するシナリオについて説明する．

KAG3 で記述したスクリプトファイルは基本的にタグとコマンドを利用して作成する．タグは[ ]の間にタグ名を記入する．たとえば，jump と呼ばれるタグであれば，[jump] と記述する．タグには属性と呼ばれるオプションがあり，属性を利用する場合は，タグ名や前の属性の後に半角スペースを入れてから，属性名＝“属性の値”を記述することで利用できる．属性の値の“”は基本的にはなくてもよいが，属性の値が半角スペースを持つ場合は必須になる[5]．たとえば [jump] のタグの属性 target を利用する場合，[jump target=start] といった形になる．また，タグにより利用できる属性の種類は異なり，タグによっては記述が必須の属性も存在する．

コマンド行は，@で始まり，その次にタグ名と属性を記述する．1 つの行には 1 つのタグのみを記述できる[5]．

また様々な種類のタグの中で，macro，endmacro，if, elsif, ignore など一部のタグを除いたタグは cond 属性と呼ばれる属性を持つ．この cond 属性には条件式を指定し，この式を評価した結果が真の時のみそのタグが実行され，偽の場合は実行されない[5]．例えば，[jump cond=f.noskip] と記述された場合，f.noskip が真の時のみ jump タグを実行する．

シナリオの基本的な動作は，ジャンプ操作でジャンプ先を指定し，条件分岐命令により様々なルートを通るものである．ジャンプ先の指定は，ラベルを用いて行う．ラベルはジャンプ先の指定の他にセーブやロードにも使われる．

図 4 に KAG3 で記述したスクリプトファイルの一部を示す．

```
[font size=40 ]　test　[resetfont][r]
[link target="kirikiri"]吉里吉里[endlink][r]
[link target=“*kag”]　KAG　[endlink][r]
@endnowait
@history output=true
@current page=fore
@jump target="*about_end"
```

図 4　KAG3 で記述したスクリプトファイル

4.4.2 Promela への変換に必要な要素

KAG3 で記述されたスクリプトファイルの内容のうち，モデル検査を行うために Promela に反映させる必要があるのは，条件分岐命令と変数宣言である．スクリプトは 4.4.1 で述べたように，タグとコマンドを用いて記述されるが，これらのうち，条件分岐に影響し Promela への変換で必要になる操作は以下に述べる 2 つである．

1 つはラベル・ジャンプ操作である．この操作は，スクリプトファイル内にあるラベルをジャンプ先としてジャンプする．ラベル・ジャンプ操作のコマンドは全部で 12 種類存在する．全 12 種類中，ノベルゲームのシナリオの条件分岐に影響を及ぼすのは, 以下の 4 種類であると考えられる．

① button

button（グラフィカルボタン）は指定された画像をクリックすることでジャンプする．16 種類の属性を持つが，条件分岐に直接影響があるのは，storage, target, exp の 3 つである．storage はジャンプ先のシナリオファイルを指定する. 省略すると現在のシナリオファイル内であるとみなす. target はジャンプ先のラベルを指定する．省略するとファイルの先頭から実行する．exp はクリックされたときに実行される式を指定する．省略すると式は実行されない．storage, target が指定されていた場合，式を指定された式を実行してからジャンプ先に移動する．この 3 つの属性は button タグを利用する際には必須ではない．

② jump

jump（シナリオのジャンプ）は指定されたシナリオファイルの指定されたラベルにジャンプする．属性は 3 種類あり，そのうち storeage, target が条件分岐に影響を及ぼす．この 2 つの属性は必須ではない．

ⓒ 2017 Information Processing Society of Japan

③　link

link（ハイパーリンク）は link タグと endlink タグで囲まれた文章を，マウスやキーボードで選択可能にし，そこでクリックやキーを押された際にジャンプする先を指定できる．属性は 14 種類あり，そのうち storeage, target, exp が条件分岐に影響を及ぼす．この 3 つの属性は link タグを利用する際には必須ではない．

④　endlink

endlink（ハイパーリンクの終了）は，link を利用する際は必須となるタグである．またこのタグには属性がない．

2 つ目の操作は，変数操作である．変数操作は，変数を代入したり，評価したりする操作である．この操作には 14 種類のタグが存在するが，シナリオの条件分岐に影響を及ぼすタグは以下の 7 種類である．

⑤　eval

eval（式の評価）は属性として exp のみを持ち，exp で示された式を評価する．通常は変数への代入に用いる．このタグを利用する際，exp は必須となる．

⑥　if

if（条件によりシナリオを実行）は式を評価し，その結果が true（または 0 以外）ならば，elsif, else, endif のいずれかのタグまでにある文章やタグを実行し，そうでない場合は無視する．このタグは exp のみを属性として持ち，タグを利用する際 exp は必須となる．

⑦　else

else（if の中身が実行されなかったときに実行）は if タグもしくは elsif タグ と endif タグの間で用いられる．if または elsif の中身がひとつも実行されていないとき，else から endif までの間を実行する．またこのタグは属性を持たない．

⑧　elsif

elsif （それまでの if の中身が実行されていなかったときに，条件付きで実行）は if タグと endif タグの間で用いられる． それまでの if タグまたは elsif タグの中身がひとつも実行されていないときに 式を評価し，その結果が真ならば elsif から次の elsif, else, endif までの間を実行する．またこのタグは属性として exp を持ち，elsif タグを利用する際には必須となる．

⑨　endif

endif（条件よりシナリオを実行の終了）は if タグの終了を示す．このタグには属性は存在しない．

⑩　ignore

ignore（条件によりシナリオを無視）は式を評価し，その結果が true（または 0 以外）ならば，ignore と endignore の間にあるタグや文章を全て無視する．また属性として exp があり，このタグを利用する際はこの属性が必須となる．

⑪　endignore

endignore（条件よりシナリオを無視の終了）は ignore タグの終了を示す．このタグには属性は存在しない．

条件分岐には影響しないが，KAG3 の構造上の問題の回避のために必要な要素のタグがある．それを以下に示す．

⑫　s

s（停止する）タグは，シナリオの停止させるために利用する．特に button や link などの選択待ちの際に利用する．このタグに属性は存在しない．このタグが必要な理由は，KAG3 で記述したシナリオはジャンプ操作がされない限り，下のラベルに進むためである．停止タグを変換しておかなければ，本来進むはずのないルートを通るシナリオの Promela に変換されてしまうからである．

先述したタグと属性に cond 属性を加えたものがシナリオの条件分岐に影響を及ぼすものとなる．

4.4.3 Promela への変換手法

シナリオのモデルはラベルで状態を管理し，goto 文と if 文を利用することで表現できる．

シナリオファイルに存在するラベルをそのまま Promela でも利用する．Promela にはラベルが存在し，goto 文でラベルの位置までジャンプすることができる．本研究でもこのラベルを利用する．Promela と KAG3 でのラベルの記述方法はほぼ同じであるため，シナリオの構造を変更することなく Promela へと変換することができる．

ここからは，4.4.2 で述べたタグや属性の変換方法について説明する．ここで説明するタグの変換では，cond 属性が影響しない場合の変換を説明する．cond 属性の変換に関しては，後述する．

②の jump タグは，goto 文に変換し，属性にある storage と target の要素からジャンプ先のラベルに変換する．図 5 に jump タグの変換例を示す．



図 5　jump タグの変換例

①の button タグと③の link タグの変換は，基本的には jump と同じように goto 文を利用するが，同じ画面に複数の button や link がある場合は，変換の方法が異なってくる．変換方法が異なる理由は，button と link はノベルゲームにおける選択肢の役割を果たすためである．ゲーム上で選択肢を選ぶのはゲームをしているプレイヤーであるため，Promela への変換では非決定性を持った if 文を利用する．Promela の if 文には，決定性と非決定性が存在し，決定性はガード条件がある if 文であり，非決定性はガード条件がなく，命令はランダムに選択される．一画面に出てくる button と link をまとめ 1 つの if 文の中に入れる．1 つ 1 つの button と link タグは jump と同じように goto 文に変換し，属性にある storage と target の要素からジャンプ先のラベルに変換する．図 6 に，一画面に button が 2 つある場

ⓒ 2017 Information Processing Society of Japan

合の変換例を示す.

KAG3

```
[button target=*label1]
[button target=*label2]
```

Promela

```
if
        :: goto label1;
        :: goto label2;
fi;
```

図 6　button タグの変換例

④の endlink は基本的には変換しなくてもよいが，link の属性が多い場合のみ，link タグの終了を探すために利用する.

⑥の if タグ，⑦の else タグ，⑧の elsif，⑨の endif の変換は, 属性の exp を基に評価式をガード条件として変換し，if 文を作成する. また if と else か elsif の間にあるタグを処理部分にして変換する. elsif と次の elsif の間のタグも同様に変換していく. endif の部分を fi に変換する. 図 7 に if, else, elsif, endif タグの変換例を示す.

KAG3

```
[if exp="x=0"]
@jump target=*test0
[elsif exp="x=1"]
@jump target=*test1
[else]
@jump target=*test2
[endif]
```

Promela

```
if
        ::(x==0)->
        goto test0;
        ::(x==1)->
        goto test1;
        :: else ->
        goto test2;
fi
```

図 7　if, else, elsif, endif の変換例

⑩の ignore タグと⑪の endignore の変換は ignore タグから endignore タグの間のタグを全て if 文の処理部分に変換し，ガード条件を ignore タグの属性 exp の値にする. またもう 1 つの処理部分に skip を，ガード条件を else にすることで変換する. 図 8 に ignore タグと endignore タグの変換例を示す.

KAG3

```
[ignore exp="x=2"]
@eval exp="f3 =10"
@eval exp="ans =2"
@eval exp="ans2 =f3+ans "
@jump target=*test1
[endignore]
```

Promela

```
if
        ::(x==2)->
        f3=10;
        ans=2;
        ans2=f3+ans;
        goto test1;
fi;
```

図 8　ignore タグの変換例

⑤の eval タグの変換は，変数の代入として変換する. ここで問題となるのが，Promela には存在する型宣言が，KAG3 には存在しないということである. eval タグの属性 exp に“変数名＝値”と記述することで変数に値を代入できるが，値が数字の場合，変数は数値として，値が文字列の場合は文字列として扱われる. そのため変換の際には，代入された値が数値か文字列かを判断しなければいけない. 変数が文字列の場合は列挙型の mtype に変換し，数値の場合は int 型に変換する. また Promela では，“.”を抜き取った変数名にする. 図 9 に eval タグの変換例を示す.

KAG3

```
@eval exp="f.3 =10“
[eval exp="f.2=‘テスト’"]
@eval exp="f.1 =f.3“
```

Promela

```
int f3=10;
mtype={テスト}
mtype f2=テスト;
int f1=f3;
```

図 9　eval タグの変換例

以上が基本的なタグの変換方法である. 次に cond 属性の変換について説明する. cond 属性は，ignore タグと同じ変換方法になる. タグに cond 属性が利用されているかどうかをチェックし，利用されている場合は，評価する式をガード条件とし，タグを変換したものを処理部分にする. そして，処理部分を skip とし，ガード条件に else を記述したものを加えることで，cond 属性は変換できる. 図 10 に cond 属性の変換例を示す.

KAG3

```
[jump target=*test1 cond="x=0"]
```

Promela

```
if
        ::(x==0)-> goto test1
        :: else -> skip;
fi;
```

図 10　cond 属性の変換例

⑫の s タグの変換は今いるラベルに戻る goto 文に変換する. この理由は Promela には停止するための stop がないためである. 停止の代わりに今いるラベルへ飛ぶことで，疑似的な停止を表現する. この s タグを変換した例が図 11 のようになる.

KAG3

```
*lavel1
@jump cond="x=1" target=*label1;
[s]
```

Promela

```
lavel1:
if
:: x=1 -> goto label1;
:: else -> skip
fi
goto lavel1;
```

図 11　s タグの変換例

4.5 典型的な不具合のパターンとその LTL 式への変換

4.5.1 典型的な不具合のパターン

シナリオに存在する典型的な不具合のパターンは，[3]によれば 4 つあり，それぞれの不具合を SPIN で検証するために必要な記述は以下のようになる.

1 つ目は，到達不可能なラベルの存在があるということである. この不具合を LTL 式にすると[]〈〉(@ラベル名)という式になる. この LTL 式は，「いつかは必ずラベル名

ⓒ 2017 Information Processing Society of Japan

になる」という意味である．

2 つ目は，エンディングに到達しない「ハマり」状態が存在することである．この不具合は SPIN が持っている機能を利用することで検証できると考えられる．SPIN には Promela で記述したモデルが正常に停止したかどうかをチェックする機能がある．これを利用するにはエンディングのラベルの名前の先頭に end と付けることが必要である．end と付いたラベルか，プロセスの最後のラベル以外で終了した場合に異常終了したラベルを反例として出力する．

3 つ目はストーリー依存関係の矛盾である．ストーリー依存関係の矛盾は，ラベル A の後にラベル B を通ることはない，という形の LTL 式になる．

4 つ目は，各種 invariant（不変）の矛盾である．これは変数 x が 1 のときラベル C を必ず通る，といったような形の LTL 式となる．

4.5.2 テンプレート

4.5.1 で説明したような LTL 式に変換するため，満たすべき性質を記述できるテンプレートを作成する．

1 つ目の到達不可能なラベルの存在用のテンプレートは，チェックするためのラベル名を記述することで検証できる形になる．このテンプレートをテンプレート A とする．テンプレート A を図 12 に示す．このテンプレートに確認したいラベル名を（）で括って記述する．確認したいラベルが複数ある場合は 1 つ目のラベルと同様に記述していけば良い．

```
(ラベル名1)(ラベル名2)
```

図 12　テンプレート A

テンプレート A に従って記述した例を図 13 に，変換後の LTL 式を図 14 に示す．図 14 に示す LTL 式は，「いつかは必ずプロセス process の test1 と test2 を通る」ということを意味する．

```
(*test1)(*test2)
```

図 13　テンプレート A の記述例

```
[]((<>(process@test1))&&(<>(process@test2)))
```

図 14　テンプレート A の変換例

2 つ目の不具合の「ハマり」の存在用のテンプレートは，作成しない．4.5.1 で説明したように「ハマり」は SPIN の機能で検証するため LTL 式を必要としないからである．

3 つ目の不具合のストーリー依存関係の矛盾用のテンプレートは，依存関係の矛盾を検証したいラベル名を記述することで検証できる形になる．このテンプレートをテンプレート B とする．テンプレート B を図 15 に示す．

```
（ラベル名1） ->（ラベル名2）
```

図 15　テンプレート B

直接的なストーリー依存関係が 2 つのラベルのみの場合には，()　の中にはラベル名を記述することで，ストーリー依存関係の矛盾を発見できるが，ラベルが 3 つ以上の場合は，そのまま“->（ラベル名 3)”と記述することで検証できる．また「ラベル 1 の後，ラベル 2 を通った場合，必ずラベル 3 の後，ラベル 4 を通る」といった依存関係を検証したい場合は，1 つ目と 2 つ目の依存関係を別々にテンプレート同様に記述し，それぞれ（）で括り，1 つ目と 2 つ目の依存関係を->で繋ぐことで検証できる．このテンプレートに従って記述した例を図 16 に，変換後の LTL 式を図 17 に示す．図 17 に示す LTL 式は，「test1 の後，いつかは必ず test2 を通る」ということを意味する．

```
((*test1)->(*test2))
```

図 16　テンプレート B の記述例

```
[]((process@test1)->(<>(process@test2)))
```

図 17　テンプレート B の変換例

4 つ目の不具合の各種 invariant（不変）の矛盾用のテンプレートは，変数名と評価式，ラベル名を記述することで検証できる形になる．このテンプレートをテンプレート C とする．テンプレート C を図 18 に示す．

```
（評価式1 or ラベル名1） ->（評価式2 or ラベル名2）
```

図 18　テンプレート C

テンプレート C に従って記述した例を図 19 に，変換後の LTL 式を図 20 に示す．図 20 に示す LTL 式は，「(x=1 かつ y=2）ならば，(test1 の後，必ず test2 を通る)」ということを意味する．

```
((x=1)&&(y=2))->((*test1)->(*test2))
```

図 19　テンプレート C の記述例

```
[](((x==1)&&(y==2))->((process@test1)->(<>(process@test2))))
```

図 20　テンプレート C の変換例

ⓒ 2017 Information Processing Society of Japan

5. 評価

5.1 評価方法

本提案ツールによってゲームシナリオが検証できることを評価するため，不具合が存在することがわかっているスクリプトファイル本ツールで検査し，不具合が発見できるかテストした．典型的な 4 種類の不具合のそれぞれについて，1 つずつを意図的に埋め込んだ 4 つのスクリプトファイルを作成し，それぞれのスクリプトファイルをツールで検証する．

テストしたゲームシナリオの流れを図 21 に示す．このシナリオに典型的な 4 種類の不具合のそれぞれについて，1 つずつを意図的に埋め込んだ．

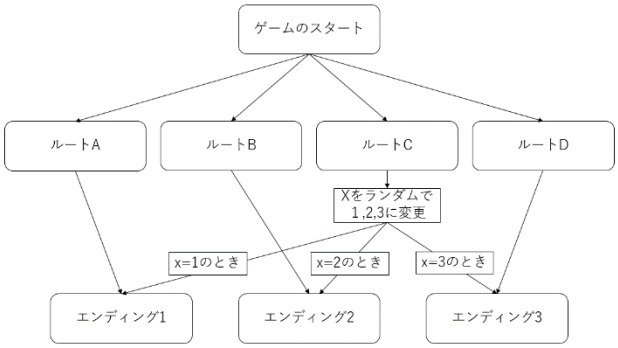


図 21　シナリオの流れ

5.2 評価結果

4 つのスクリプトファイルを検証した結果，不具合を検出することができた．

到達不可能なラベルの存在を持つスクリプトファイルから，エンディング 3 のラベルが存在するのにも関わらず，エンディング 3 の状態に到達しないことを検証できた．

到達不可能なラベルの存在を持ったスクリプトファイルから，同一の状態で止まってしまう「ハマり」状態を発見することができた．

また，ストーリー依存関係の矛盾も発見できた．

さらに，各種 invariant（不変）の矛盾を持ったスクリプトファイルから,仕様を満たしていない部分を発見できた．

5.3 考察

複数の不具合が存在する場合は,「ハマり」状態の存在が先に検出される．「ハマり」状態が先に発見される理由は，end ラベル以外の部分で終了を起こしているエラーから検出してしまうためである. そのため出力結果は「self-loop が存在する」のみとなる．この結果や不具合の性質から，全ての不具合を発見するためには,「ハマり」状態の存在，到達不可能なラベルの存在，ストーリー依存関係の矛盾，各種 invariant（不変）の矛盾の不具合の順に検証するべきだと考える．

ⓒ 2017 Information Processing Society of Japan

6. おわりに

本稿では，モデル検査の知識を持たないゲーム製作者でも利用できる，シナリオの不具合発見ツールを提案した．本ツールは，検証対象を条件分岐命令によってストーリーが異なるノベルゲームに絞り，シナリオを記述したスクリプトファイルを SPIN のモデル記述言語 Promela へと変換することでモデル検査を行う．また，シナリオの典型的な不具合のパターンから満たすべき性質を記述するためのテンプレートを定めることにより，LTL 式の知識がなくても本ツールを利用して性質を満たすかを検査できるようになった．

今後の課題として，ゲームのシナリオだけでなく，ゲームの他の機能の不具合の発見にも適用できるようなツールの作成がある．本稿で提案したツールを基に，他の機能を検証するためのモデルへの変換と他の典型的な不具合のパターンを見つけ出すことで，シナリオ以外の機能にも適用できるのではないかと考える．ゲーム開発は製作者によって細かい製作方法が異なるために，シナリオ以外の他の機能にモデル検査を適用するためには，典型的な製作方法を調査し，その方法毎の変換方法を検討する等が必要になると思われる．

参考文献

[1] Clarke, E.M, et.al: Model Checking, MIT Press (1999)
[2] Holzmann, G.J..: The Model Checker SPIN. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol.23, No.5 (1997)
[3] 清木昌．ゲームシナリオのモデル検査．情報処理学会 研究報告 ゲーム情報学, vol. 28, p. 51-56. (2004)
[4] “吉里吉里 Z”. http://krkrz.github.io/ (参照 2017-01-05)
[5] “KAG System リファレンス -kikyo.info-/タグリファレンス”. http://devdoc.kikyou.info/tvp/docs/kag3doc/contents/.,（参照 2017-01-05）

ⓒ 2017 Information Processing Society of Japan