Vol.2021-SE-209 No.4
2021/11/25

UE4 Blueprint で作成されたゲームプログラムのモデル検査手法の提案

和山 一樹[1,a)]　横川 智教[,b)]　井川 直[1,c)]　有本 和民[1,d)]

概要：本論文では，Unreal Engine4 Blueprint（以下，Blueprint）で作成されたゲームプログラムに対してモデル検査を適用するためのモデル化手法について検討を行っている．Blueprint では様々な処理を行うノードを組み合わせることでゲームロジックを視覚的に記述できるが，その規模が大きくなるにつれて，ゲームの進行不可バグなどの不具合を発見・修正することが困難になる．そこで本論文では，形式検証技術であるモデル検査を用いて，ゲームロジックのバグを自動的に検出する．本論文では，Blueprint で作成されたゲームプログラムをモデル検査ツール NuSMV の入力モデルへと変換することで，NuSMV による検証を実現する．適用実験として，Blueprint で作成したゲームプログラムに提案手法を適用してモデル化および NuSMV による検証を行っている．

1. まえがき

近年，ゲーム開発の分野においてビジュアルスクリプティングシステムである UE4 Blueprint [4] が広く用いられている．UE4 Blueprint では変数や関数などがノードと呼ばれる視覚的なブロックとして扱われ，それらのノードを組み合わせることでゲームロジックを表現することが可能である．ここで，ゲーム内におけるキャラクターの動きや衝突判定などのゲームロジックを表現するノードのまとまりをブループリントと呼ぶ．ゲームの規模が大きくなるにつれて，複雑なブループリントを多数作成することとなり，単純な不具合であってもその要因を発見・修正することが困難になる．こういった問題を解決する有効な手法として，システムの自動検証技術であるモデル検査 [3] が注目されている [5][7][8].

本研究では，UE4 Blueprint で作成されたゲームプログラムに関して，モデル検査を適用するための手法を提案する．ブループリントのモデル化にあたって，フローのふるまいを表現するとともに，モデルの最適化の実現を目指す．参考書を用いて作成したゲームプログラムを対象として，様々なケースについてのブループリントを実際にモデル化した．また，モデル検査ツールとしては，高速な記号モデル検査アルゴリズムを実装したツールである NuSMV [2] を用いている．

[1] 岡山県立大学, 岡山県総社市窪木 111, Okayama Prefectural University, 111 Kuboki, Soja, Okayama 719–1197, Japan
[a)] sk621053@cse.oka-pu.ac.jp
[b)] t-yokoga@cse.oka-pu.ac.jp
[c)] igawa.nao.opu@technoaccel.com
[d)] arimoto@cse.oka-pu.ac.jp

2. 準備

2.1 モデル検査

モデル検査は有限状態のコンピュータシステムを対象とした形式的検証技術である．数学や論理学を基礎として対象システムの状態遷移構造を記号的に表現し，状態探索によって対象システムが検証特性を満たすか否かを判定する．検証特性を満たさないと判定された場合，その特性を満たさない状態までの状態系列（すなわちシステムの実行系列）が反例として出力される．なお，検証特性は時相論理と呼ばれる論理を用いて表す．時相論理は，時間の要素を明示的に表現することなく，時間を伴うイベントの順序を記述可能な論理である．本研究では，線形時相論理 LTL を用いて検証特性を表現する．

2.2 NuSMV

NuSMV [2] は記号モデル検査の実装の一つである．NuSMV で検証されるモデルは，SMV 言語と呼ばれる専用の入力言語で記述される．検査対象となる特性は，LTL (Linear Temporal Logic) [6] や CTL (Computational Tree Logic) [1] などの時相論理で表現される．以下に，NuSMV への入力モデルの例を示す．

```
1 MODULE main
2 VAR
3   sw : {on, off};
4 ASSIGN
5   init(sw) := {on, off};
```

ⓒ 2021 Information Processing Society of Japan

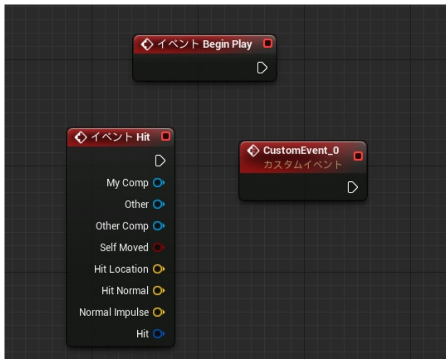

図 1 イベントノード
Fig. 1 Event Nodes

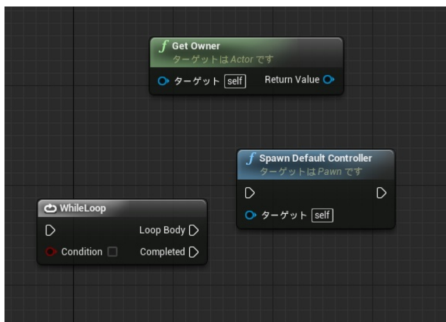

図 2 関数・マクロノード
Fig. 2 Function/Macro Nodes

```
 6    next(sw) := case
 7      sw = on : off;
 8      TRUE : sw;
 9    esac;
10  CTLSPEC AG (AF sw = on)
```

SMV 言語で記述された入力モデルは，VAR で記述される変数宣言部と ASSIGN で記述される遷移定義部から構成される．検査する特性となる LTL 式および CTL 式は，LTLSPEC および CTLSPEC を用いて入力モデルへと注釈づけられる．

2.3 UE4 Blueprint

UE4 Blueprint は，Unreal Engine 4 のビジュアルスクリプティングシステムであり，プレイヤーや敵キャラクターの行動，スコア計算，ゲームの終了条件などのゲームロジックをノードの接続によって視覚的に記述する．Blueprint スクリプティングシステムでは，C 言語などのプログラミング言語で使用される変数や関数などの要素すべてがノードとして視覚的に扱うことができる．そのため，プログラミングの専門知識を有しないユーザであっても，ゲームロジックを直感的に記述することが可能である．

ノードの接続によりゲームロジックの特定の振る舞いを表現したひとまとまりのグラフをブループリントと呼ぶ．ゲームの規模や複雑さに応じてブループリントの数や，含まれるノードの種類，数は異なる．

ブループリントはノードとワイヤから構成される．ノー

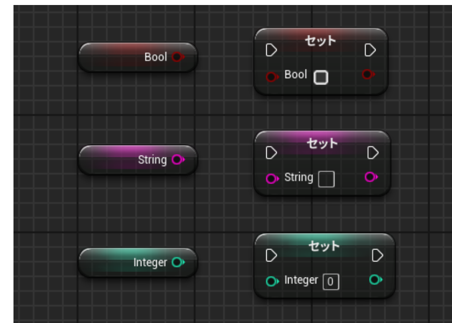

図 3 変数ノード
Fig. 3 Variable Nodes

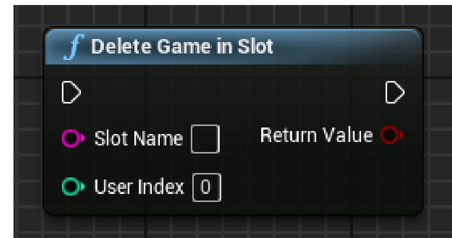

図 4 ピンの種類
Fig. 4 Pin types

ドは，変数などのゲームの要素やゲーム内の処理を表す関数，また，定義済みで再利用可能なブループリントを表す．ワイヤは，ピンを介してノード間を接続し，データや処理の流れを表現する．ピンは，データの流れを表すワイヤを接続するためのデータピンと，処理の流れを表すワイヤを接続するための実行ピンに分類される．

本研究で扱うノードは，イベントノード，関数・マクロノード，変数ノードの 3 種である．イベントノード（赤色）は，ブループリントの実行の起点となるノードである．図 1 にイベントノードの例を示す．

関数ノードは，プログラム中の値に対する関数を実装したノードである．関数ノードは，non-pure 関数ノード（青色）と pure 関数ノード（緑色）に分類される．non-pure 関数は実行ピンにより他のノードと処理が同期される関数で，pure 関数は実行ピンをもたず，データが入力された時点で処理を行う関数である．マクロノード（灰色）は，関数ノードと同様に定義済みの処理を表しており，再利用可能なブループリントを実装したノードである．図 2 に関数ノードとマクロノードの例を示す．

変数ノードは，オブジェクトやアクタの値を保持するための変数を表すノードである．変数の型は，Boolean 型，Integer 型，Float 型，String 型，あるいは他のオブジェクトへの参照などがあり，型に応じてノードの色が異なる．変数は，get ノードあるいは set ノードとしてブループリント上に配置される．get ノードは変数の値を参照するために使用される．set ノードは変数の値を更新するために使用され，実行ピンによって呼び出される．図 3 に変数ノードの例を示す．

ⓒ 2021 Information Processing Society of Japan

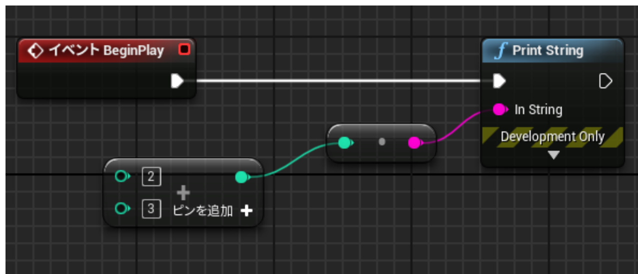

図 5　2+3 の結果を画面に表示するブループリント
Fig. 5　Blueprint to display the result of 2+3 on the screen

ピンは，上述の通り実行ピンとデータピンに分類される．ノードの左側に配置されたピンが入力，右側に配置されたピンが出力に用いられる．図 4 にピンの例を示す．実行ピンはノードの処理を他のノードと同期するために用いられるもので，白色で表される．入力実行ピンの接続先ノードの処理が完了したとき，ノードの処理が開始される．そして，処理が完了した後，出力実行ピンの接続先ノードの処理が開始される．データピンはノードへのデータ入出力に用いられるもので，データの型に応じた色で表される．図 4 の例では Slot Name ピンは String 型であり，User Index ピンは Integer 型の入力ピンである．Return Value ピンは Boolean 型の出力ピンである．

ワイヤは，実行ピン間，あるいはデータピン間のみを接続する．また，データピンは同一の型をもつピン間の接続のみが許される．実行ピン間は白色のワイヤで接続され，これらのワイヤが表す一連の実行の流れを実行フローという．データピン間は変数の型に応じた色のワイヤで接続され，これらのワイヤが表す一連のデータの流れをデータフローという．

3.　ブループリントのモデル検査

本研究では，ブループリントの各ノードの入出力ピンに SMV プログラムの変数（以下，SMV 変数という）を割り当てた上でそれらの振る舞いを個別に記述することでフローを表現し，ブループリントを SMV プログラムとしてモデル化する．検査対象となるゲームプログラムを構成するすべてのブループリントをモデル化し，合成することでゲーム全体のモデルを得ることができる．以下，図 5 のブループリントを例として，SMV プログラムへとモデル化する手順について述べる．このブループリントは，実行の起点となるイベントノード BeginPlay，定数 2 と 3 の加算を行う pure 関数ノード Plus，整数から文字列への変換を行う pure 関数ノード ToString，そして文字列を出力する non-pure 関数ノード PrintString から構成されている．

3.1　フローのモデル化

データフローをモデル化するためには，どのノードからピンとワイヤを介してどのようなデータが遷移してきたかを表現する必要があるため，各ノードのデータピンごとに SMV 変数を割り当てる．ここで，定義する SMV 変数の型は対応するデータピンの型によって異なる．例えば，データピンが Boolean 型であった場合は SMV 変数も Boolean 型とし，Integer 型であった場合は SMV 変数も整数型とする．SMV 言語では整数型の変数に有限の定義域を割り当てる必要があるため，ノードの振る舞いを表現するのに十分な範囲を割り当てる．String 型の場合，SMV 言語では文字列を扱えないため，列挙型として定義する．図 5 の例のデータフローに対して定義された SMV 変数を以下に示す．

```
1  VAR
2    Plus_ToString : 0..5;
3    Plus_ToString_defined : boolean;
4    ToString_PrintString : {_Undefined, is5, not5};
```

Plus の出力データピンおよび ToString の入力データピンは Integer 型で 5 までの値をとるので定義域 0..5 の整数型変数として定義し，ToString の出力データピンおよび PrintString の入力データピンは String 型で値 5 をもちうるので定義域としては値 5 とそれ以外の値をとる列挙型として定義する．ここで，データフローに関してはワイヤを介して接続された 2 つのピンは同じ値をもつことになるため，1 つの変数を割り当てれば十分である．したがって，この例では Plus の出力データピンと ToString の入力データピン間のワイヤと ToString の出力データピンと PrintString の入力データピン間のワイヤに変数を割り当てる．なお，ブループリントの処理の開始時点など，変数の値が定義されていない状態が存在しうる．このような状態であることを，値 `Undefined` として表現する．整数型の変数については，シンボルと数値を一つの定義域に含めることができないため，未定義であることを表すフラグ変数を新たに定義している．

実行フローをモデル化するためには，次にどのノードを実行するかを表現する必要がある．実行フローが 1 つのみ，すなわち入出力実行ピンを複数もつようなノードが存在しない場合は，実行フローを表すための SMV 変数を 1 つ定義すればよい．このとき変数の型は，実行フロー上の各ノードを表すシンボルを定義域とする列挙型となる．複数の入出力実行ピンによって，実行フローが分岐あるいは合流する場合は，分岐・合流数に応じて SMV 変数を複数定義する．図 5 の例の実行フローに対して定義された SMV 変数を以下に示す．

```
1  VAR
2    _ExecutionFlow : {_Undefined, BeginPlay, PrintString};
```

実行ピンで接続されたノードは BeginPlay と PrintString の 2 つのみであるため，これらを表すシンボルの集合が定義域となっている．

ⓒ 2021 Information Processing Society of Japan

3.2 ノードの振る舞いのモデル化

次に，それぞれのノードの振る舞いをモデル化する．ノードの振る舞いは，入力ピンの値に応じた，出力データピンの値の変化として表現できる．図 5 の例では，Plus ノードと ToString ノードの 2 つが出力データピンをもち，上述の 3 つの SMV 変数で表現されている．これらの変数の振る舞いは，以下の SMV プログラムによって定義される．

```
1  ASSIGN
2    init(Plus_ToString) := 0;
3    next(Plus_ToString) := 2 + 3;
4
5    init(Plus_ToString_defined) := FALSE;
6    next(Plus_ToString_defined) := TRUE;
7
8    init(ToString_PrintString) := _Undefined;
9    next(ToString_PrintString) := case
10     !Plus_ToString_defined : _Undefined;
11     Plus_ToString = 5 : is5;
12     Plus_ToString != 5 : not5;
13     TRUE : ToString_PrintString;
14   esac;
```

Plus ノードの出力データピンの値は，ブループリントの実行開始時は未定義で，実行されると 2+3 の演算結果となる．ToString ノードは pure 関数なので，入力データピンの値が確定した時点で処理が開始され，出力データピンの値は，入力データピンの値に応じて 5 か 5 以外の値となる．

実行ピンについては，実行フローを表す SMV 変数 ExecutionFlow の値の変化として表現できる．図 5 の例では実行フローは 1 つのみであるため，以下の SMV プログラムとして定義できる．

```
1  ASSIGN
2    init(ExecutionFlow) := Undefined;
3    next(ExecutionFlow) := case
4      ExecutionFlow = Undefined : BeginPlay;
5      ExecutionFlow = BeginPlay : PrintString;
6      TRUE : ExecutionFlow;
7    esac;
```

以上より，図 5 のブループリントは以下の SMV プログラムとしてモデル化できる．

```
1  MODULE main
2  VAR
3    _ExecutionFlow : {_Undefined, BeginPlay, PrintString};
4    Plus_ToString : 0..5;
5    Plus_ToString_defined : boolean;
6    ToString_PrintString : {_Undefined, is5, not5};
7    Output_PrintString : {_Undefined, is5, not5};
8
9  ASSIGN
10   init(_ExecutionFlow) := _Undefined;
11   next(_ExecutionFlow) := case
12     _ExecutionFlow = _Undefined : BeginPlay;
13     _ExecutionFlow = BeginPlay : PrintString;
14     TRUE : _ExecutionFlow;
15   esac;
16
17   init(Plus_ToString) := 0;
18   next(Plus_ToString) := 2 + 3;
19
20   init(Plus_ToString_defined) := FALSE;
21   next(Plus_ToString_defined) := TRUE;
22
23   init(ToString_PrintString) := _Undefined;
24   next(ToString_PrintString) := case
25     !Plus_ToString_defined : _Undefined;
26     Plus_ToString = 5 : is5;
27     Plus_ToString != 5 : not5;
28     TRUE : ToString_PrintString;
29   esac;
30
31   init(Output_PrintString) := _Undefined;
32   next(Output_PrintString) := ToString_PrintString;
33
34 LTLSPEC !F (Output_PrintString = is5)
```

PrintString ノードによる画面への出力を SMV 変数 Output_PrintString としてモデル化している．そして，検査特性として「いつか画面への出力が 5 となることはない」という LTL 式 !F (Output_PrintString = is5) を与えている．

この SMV プログラムを NuSMV で検証した結果は以下の通りとなる．

```
1  -- specification !( F Output_PrintString = is5) is false
2  -- as demonstrated by the following execution sequence
3  Trace Description: LTL Counterexample
4  Trace Type: Counterexample
5    -> State: 1.1 <-
6      _ExecutionFlow = _Undefined
7      Plus_ToString = 0
8      Plus_ToString_defined = FALSE
9      ToString_PrintString = _Undefined
10     Output_PrintString = _Undefined
11   -> State: 1.2 <-
12     _ExecutionFlow = BeginPlay
13     Plus_ToString = 5
14     Plus_ToString_defined = TRUE
15   -> State: 1.3 <-
16     _ExecutionFlow = PrintString
17     ToString_PrintString = is5
18   -- Loop starts here
19   -> State: 1.4 <-
20     Output_PrintString = is5
21   -- Loop starts here
22   -> State: 1.5 <-
23   -> State: 1.6 <-
```

上述の LTL 式は偽となり，「いつか画面への出力が 5 となる」ことが示されるとともに，反例として画面出力が 5 となるまでの遷移系列が出力されている．

3.3 定義域の抽象化による状態空間削減

大規模なゲームロジックは多数のブループリントから構成されるため，モデル検査を適用した場合に状態数が爆発的に増加して，現実的な時間での検証が困難となる恐れがある．状態数に影響を与える要因としては，SMV 変数の個数に影響を与えるノード数や各ノードのピン数に加えて，SMV 変数の定義域のサイズが挙げられる．特にゲームプログラムでは，「キャラクターが元々いた座標と移動後の座標からベクトルの長さを計算し，キャラクターのスピードを求める」などのように複雑な計算を行うノードが多く，こういったノードの振る舞いをそのままモデル化すると，各変数の定義域の大きさから状態爆発が生じる危険性が大きい．例えば，上記のノードに関連するデータフローをモデル化するためには，以下のような SMV 変数を定義する必要がある．

```
1  VAR
2    x1 : 1..100;
3    x2 : 1..100;
4    y1 : 1..100;
5    y2 : 1..100;
6    x1_defined : boolean;
7    x2_defined : boolean;
8    y1_defined : boolean;
9    y2_defined : boolean;
10   speed : 0..100;
11   speed_defined : boolean;
```

このように，ゲーム内空間のサイズに応じて状態数が増加しており，こういったノードが増加するごとに指数的に状態空間が大きくなる．しかし，検証したい性質がデータではなく実行フローに大きく依存する場合は，データフローの内容を厳密にモデル化しなくても求める性質を検証することが可能である．そこで本研究では，モデル化の際に変数の定義域を抽象化することで状態数を削減し，検証コストを低減する．具体的には，数値計算を行うノードについて，計算対象および計算結果の値や計算処理について直接モデル化することを避けることで，状態数の削減を行う．前述の例では，以下のように抽象化を行う．

```
1  VAR
2    x1_defined : boolean;
3    x2_defined : boolean;
4    y1_defined : boolean;
5    y2_defined : boolean;
6    speed : {_Undefined, Low, Mid, High};
7
8  ASSIGN
9    init(spped) := _Undefined;
10   next(speed) := case
11     x1_defined & x2_defined & y1_defined & y2_defined :
12       {Low, Mid, High};
13     TRUE : speed;
14   esac;
```

入力となる座標値については，定義されているか否かのみを参照し，計算結果である速度は非決定的に低い（Low），中間（Mid），高い（High）のいずれかが選択されるようモデル化されている．

4. 適用例

本章では，提案したモデル化手法を用いて，検査対象としたゲームロジックのブループリントを実際にモデル化し，NuSMV による検証を行った結果について述べる．

4.1 例題ブループリント

本研究では例題として，文献 [9][10] に掲載された AddScore，PNGateActor，そして PNPawnRabbit の 3 つのブループリントに対して提案法に基づくモデル化を行い，NuSMV による検証を行った．それぞれのブループリントを図 6，図 7，そして図 8 にそれぞれ示す．AddScore は，プレイヤーキャラクターが指定されたオブジェクトを取得した際にスコアに 1 ずつ加算する，という処理を記述したブループリントである．PNGateActor は，プレイヤーキャラクターがスイッチを押したとき，効果音が鳴り対応する扉が開く，という処理を記述したブループリントである．そして PNPawnRabbit は，AI がステージ上を徘徊し，プレイヤーキャラクターを見つけると追いかけてくる，という処理を記述したブループリントである．それぞれのブループリントのサイズは以下の通りである．

- AddScore
  - ノード数：7
  - ピン数：20
  - ワイヤ数：7
- PNGateActor
  - ノード数：11
  - ピン数：42
  - ワイヤ数：13
- PNPawnRabbit
  - ノード数：32
  - ピン数：121
  - ワイヤ数：43

ここでは，PNGateActor のブループリントのモデル化について説明する．

イベントノード Open および Close はカスタムイベントと呼ばれ，対応した他のブループリントが条件を満たすことで呼び出しが行われる．GateTimeline ノードでは，カスタムイベントから受け取ったデータをもとに扉の開閉を行う．Update 実行ピンは，タイムラインの再生中は常に実行される．Direction データピンは，タイムラインの再生方向に応じて順再生 (0) か逆再生 (1) のいずれかデータが出力されている．このデータは ToInteger ノードによって Integer 型へと変換され，PlaySE ノードへ入力される．また，Left データピンには，3 次元座標上のアニメーションを表現するための Vector 型の座標情報が出力されている．Left ピンには左側の扉の 3 次元座標データのみが格納されているため，Reverse ノードにより Y 座標を反転させ，右扉の処理を実現している．SetRelativeLocation ノードでは，入力されたデータをもとに扉の位置を更新する．PlaySE ノードは，SE タイプに対応したサウンドを流す役割をもつ．

提案手法に基づいて，PNGateActor ブループリントのモデル化を行う．まず，SMV 変数は以下のように定義できる．

```
1  VAR
2    _ExecutionFlow1 : {_Undefined, Open, GateTimeline};
3    _ExecutionFlow2 : {_Undefined, GateTimeline,
         SetRelevantLocation1, SetRelevantLocation2};
4    _ExecutionFlow3 : {_Undefined, Close, PlaySE1,
         GateTimeline};
5    _ExecutionFlow4 : {_Undefined, GateTimeline, PlaySE2};
6
7    GateTimeline_SetRelevantLocation1_defined : boolean;
8    GateTimeline_Reverse_defined : boolean;
9    GateTimeline_ToInteger : 0..1;
10   GateTimeline_ToInteger_defined : boolean;
11   Reverse_SetRelevantLocation2_defined : boolean;
12   ToInteger_PlaySE2 : 0..1;
13   ToInteger_PlaySE2_defined : boolean;
```

_ExecutionFlow1 〜 _ExecutionFlow4 は実行フローを制御する変数である．_ExecutionFlow1 で Open ノードから GateTimeline ノードへの実行，_ExecutionFlow2 で GateTimeline ノードから 2 つの SetRelevantLocation ノードへの実行を表す．_ExecutionFlow3 で Close ノードから PlaySE および GateTimeline ノードへの実行，_ExecutionFlow4 で GateTimeline ノードから PlaySE ノードへの実行を表す．次に，データピン間を接続するワイヤの状態を表すための変数を定義する．まず，GateTimeline ノードと 1

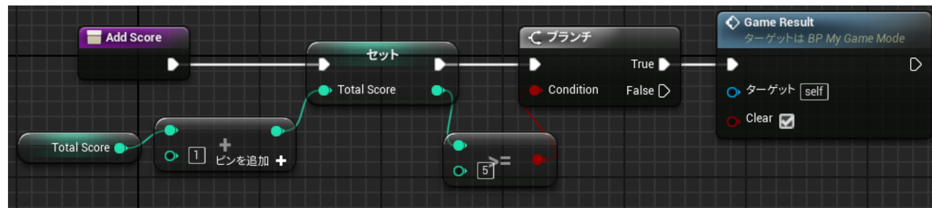

図 6 AddScore ブループリント

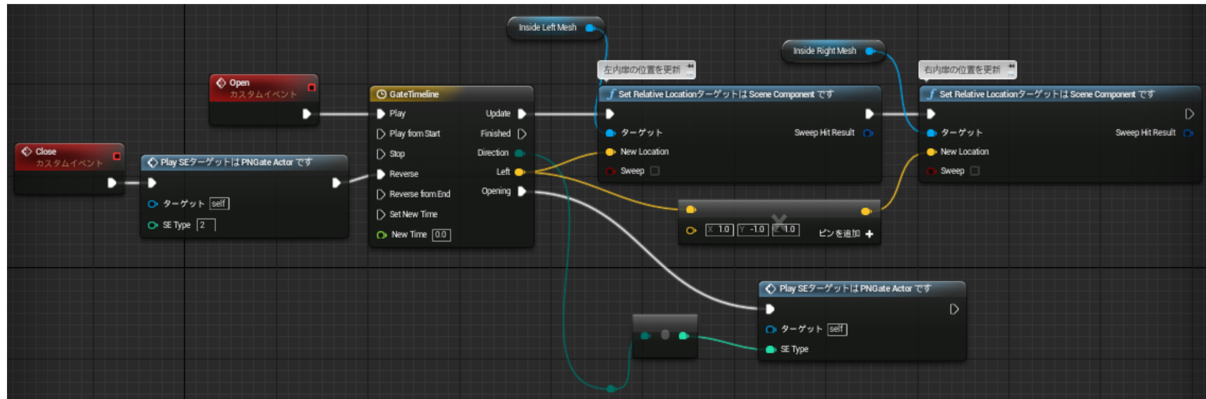

図 7 PNGateActor ブループリント

つめの SetRelevantLocation の間のワイヤについて，このブループリントでは Vector 型のデータが送受信されている．このワイヤについては抽象化を行い，Boolean 変数 GateTimeline_SetRelevantLocation1_defined を割り当てている．GateTimeline ノードと Reverse ノード間のワイヤ，Reverse ノードと 2 つめの SetRelevantLocation ノード間のワイヤについても同様である．GateTimeline ノードと ToInteger ノード間のワイヤ，ToInteger ノードと 2 つめの PlaySE ノード間のワイヤについては 0..1 を定義域とする整数として定義する．

実行フローの振る舞いは以下のように，変数 _ExecutionFlow1 〜 _ExecutionFlow4 の値の遷移としてモデル化できる．

```
1   ASSIGN
2     init(_ExecutionFlow1) := _Undefined;
3     next(_ExecutionFlow1) := case
4       _ExecutionFlow1 = _Undefined : Open;
5       _ExecutionFlow1 = Open : GateTimeline;
6       _ExecutionFlow1 = GateTimeline & _ExecutionFlow3 =
            GateTimeline : _Undefined;
7       TRUE : _ExecutionFlow1;
8     esac;
9
10    init(_ExecutionFlow2) := _Undefined;
11    next(_ExecutionFlow2) := case
12      _ExecutionFlow2 = _Undefined : GateTimeline;
13      _ExecutionFlow1 = GateTimeline & _ExecutionFlow2 =
            GateTimeline & _ExecutionFlow3 = GateTimeline
14        : SetRelevantLocation1;
15      _ExecutionFlow2 = SetRelevantLocation1 :
            SetRelevantLocation2;
16      TRUE : _ExecutionFlow2;
17    esac;
18
19    init(_ExecutionFlow3) := _Undefined;
20    next(_ExecutionFlow3) := case
21      _ExecutionFlow3 = _Undefined : Close;
22      _ExecutionFlow3 = Close : PlaySE1;
23      _ExecutionFlow3 = PlaySE1 : GateTimeline;
24      _ExecutionFlow1 = GateTimeline & _ExecutionFlow3 =
            GateTimeline : _Undefined;
25      TRUE : _ExecutionFlow3;
26    esac;
27
28    init(_ExecutionFlow4) := _Undefined;
29    next(_ExecutionFlow4) := case
30      _ExecutionFlow4 = _Undefined : GateTimeline;
31      _ExecutionFlow1 = GateTimeline & _ExecutionFlow3 =
            GateTimeline & _ExecutionFlow4 = GateTimeline
32        : PlaySE2;
33      TRUE : _ExecutionFlow4;
34    esac;
```

変数 _ExecutionFlow1 は値 Open から値 GateTimeline へと遷移し，_ExecutionFlow3 は値 Close から値 PlaySE1，GateTimeline と遷移する．_ExecutionFlow1 と _ExecutionFlow3 の双方の値が GateTimeline となるとどちらも値は未定義値 _Undefined となり，_ExecutionFlow2 と _ExecutionFlow4 に実行フローを引き継ぐ．変数 _ExecutionFlow2 は，_ExecutionFlow1 と _ExecutionFlow3 の値がどちらも GateTimeline となると実行フローを開始し，値 SetRelevantLocation1 から SetRelevantLocation2 へと遷移する．変数 _ExecutionFlow4 も同様である．

データフローの振る舞いは，以下のようにモデル化できる．

```
1   ASSIGN
2     init(GateTimeline_SetRelevantLocation1_defined) :=
            FALSE;
3     next(GateTimeline_SetRelevantLocation1_defined) :=
            case
4       _ExecutionFlow1 = GateTimeline & _ExecutionFlow3 =
            GateTimeline : TRUE;
5       TRUE : GateTimeline_SetRelevantLocation1_defined;
6     esac;
7
8     init(GateTimeline_Reverse_defined) := FALSE;
9     next(GateTimeline_Reverse_defined) := case
10      _ExecutionFlow1 = GateTimeline & _ExecutionFlow3 =
            GateTimeline : TRUE;
11      TRUE : GateTimeline_Reverse_defined;
12    esac;
13
14    init(GateTimeline_ToInteger) := 0;
15    next(GateTimeline_ToInteger) := case
```

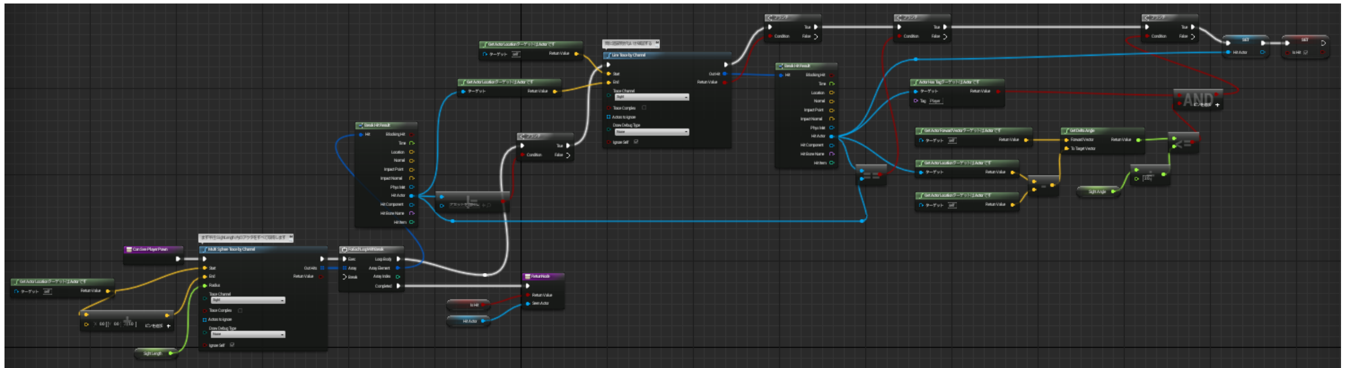

図 8 PNPawaRabbit ブループリント

```
16     _ExecutionFlow1 = GateTimeline & _ExecutionFlow3 =
           GateTimeline : 0..1;
17     TRUE : GateTimeline_ToInteger;
18   esac;
19
20   init(GateTimeline_ToInteger_defined) := FALSE;
21   next(GateTimeline_ToInteger_defined) := case
22     _ExecutionFlow1 = GateTimeline & _ExecutionFlow3 =
           GateTimeline : TRUE;
23     TRUE : GateTimeline_ToInteger_defined;
24   esac;
25
26   init(Reverse_SetRelevantLocation2_defined) := FALSE;
27   next(Reverse_SetRelevantLocation2_defined) := case
28     GateTimeline_Reverse_defined : TRUE;
29     TRUE : Reverse_SetRelevantLocation2_defined;
30   esac;
31
32   init(ToInteger_PlaySE2) := 0;
33   next(ToInteger_PlaySE2) := case
34     GateTimeline_ToInteger_defined :
           GateTimeline_ToInteger;
35     TRUE : ToInteger_PlaySE2;
36   esac;
37
38   init(ToInteger_PlaySE2_defined) := FALSE;
39   next(ToInteger_PlaySE2_defined) := case
40     GateTimeline_ToInteger_defined : TRUE;
41     TRUE : ToInteger_PlaySE2_defined;
42   esac;
```

例として，GateTimeline ノードから Reverse ノードへのワイヤならびに Reverse ノードから SetRelevantLocation ノードへのワイヤについて説明する．GateTimeline ノードから Reverse ノードへのワイヤを表す変数 GateTimeline_Reverse_defined の値は，実行フローが GateTimeline に到達した（すなわち，_ExecutionFlow1 = GateTimeline & _ExecutionFlow3 = GateTimeline のとき TRUE となる．そして，Reverse ノードから SetRelevantLocation ノードへのワイヤを表す変数 の値は，GateTimeline ノードから Reverse ノードへのワイヤの値が確定した (すなわち，GateTimeline_Reverse_defined = TRUE となった）時点で TRUE となる．

最後に検査特性については，「いつか Reverse ノードから SetRelevantLocation ノードへのワイヤに値が割り当てられることはない」ことを表す以下の LTL 式を与えている．

```
1  LTLSPEC !F(Reverse_SetRelevantLocation2_defined)
```

NuSMV による検証結果は以下の通りとなる．

```
1  -- specification !( F
       Reverse_SetRelevantLocation2_defined) is false
2  -- as demonstrated by the following execution sequence
3  Trace Description: LTL Counterexample
4  Trace Type: Counterexample
5    -> State: 1.1 <-
6      _ExecutionFlow1 = _Undefined
7      _ExecutionFlow2 = _Undefined
8      _ExecutionFlow3 = _Undefined
9      _ExecutionFlow4 = _Undefined
10     GateTimeline_SetRelevantLocation1_defined = FALSE
11     GateTimeline_Reverse_defined = FALSE
12     GateTimeline_ToInteger = 0
13     GateTimeline_ToInteger_defined = FALSE
14     Reverse_SetRelevantLocation2_defined = FALSE
15     ToInteger_PlaySE2 = 0
16     ToInteger_PlaySE2_defined = FALSE
17   -> State: 1.2 <-
18     _ExecutionFlow1 = Open
19     _ExecutionFlow2 = GateTimeline
20     _ExecutionFlow3 = Close
21     _ExecutionFlow4 = GateTimeline
22   -> State: 1.3 <-
23     _ExecutionFlow1 = GateTimeline
24     _ExecutionFlow3 = PlaySE1
25   -> State: 1.4 <-
26     _ExecutionFlow3 = GateTimeline
27   -> State: 1.5 <-
28     _ExecutionFlow1 = _Undefined
29     _ExecutionFlow2 = SetRelevantLocation1
30     _ExecutionFlow3 = _Undefined
31     _ExecutionFlow4 = PlaySE2
32     GateTimeline_SetRelevantLocation1_defined = TRUE
33     GateTimeline_Reverse_defined = TRUE
34     GateTimeline_ToInteger = 1
35     GateTimeline_ToInteger_defined = TRUE
36   -> State: 1.6 <-
37     _ExecutionFlow1 = Open
38     _ExecutionFlow2 = SetRelevantLocation2
39     _ExecutionFlow3 = Close
40     Reverse_SetRelevantLocation2_defined = TRUE
41     ToInteger_PlaySE2 = 1
42     ToInteger_PlaySE2_defined = TRUE
43   -- Loop starts here
44   -> State: 1.7 <-
45     _ExecutionFlow1 = GateTimeline
46     _ExecutionFlow3 = PlaySE1
47   -> State: 1.8 <-
48     _ExecutionFlow3 = GateTimeline
49   -> State: 1.9 <-
50     _ExecutionFlow1 = _Undefined
51     _ExecutionFlow3 = _Undefined
52   -> State: 1.10 <-
53     _ExecutionFlow1 = Open
54     _ExecutionFlow3 = Close
55   -> State: 1.11 <-
56     _ExecutionFlow1 = GateTimeline
57     _ExecutionFlow3 = PlaySE1
```

SetRelevantLocation ノードへのワイヤにも値が割り当てられることが検証でき，反例としてその状態へと至る遷移系列を得ることができた．抽象化により，データワイヤの具体的な値はモデルから削除されるが，実行フローに伴うデータの流れは表現することができている．

3 つのブループリントをモデル化した SMV プログラムに対して，NuSMV を用いて検証するために要した時間は AddScore ブループリントは 66.67 ミリ秒，PNGateActor

ブループリントは 58.13 ミリ秒，PNPawnRabbit ブループリントは 111 ミリ秒であった．検証を行った計算機環境は以下の通りである．

- OS：Windows 10
- プロセッサ：Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz 3.60 GHz
- メモリ：16.0 GB
- NuSMV：NuSMV-2.6.0-win64

抽象化により，非常に短い時間での検証が可能となっている．

5. まとめ

本研究では，UE4 Blueprint で作成されたゲームプログラムを対象として，モデル検査による検証を行うための手法を開発した．モデル検査ツールとしては NuSMV を用いており，検査対象となるブループリントを，NuSMV の入力言語である SMV 言語でモデル化するための手法を示した．適用実験として，UE4 Blueprint で作成された 3 つのブループリントを対象として，提案手法に基づいてモデルを作成した．作成したモデルに対して NuSMV を用いた検証を行い，提案手法によってブループリントが正しくモデル化されていることを確認し，短時間での検証が可能であることを確認した．今後の課題として，ゲームプログラムのバグ検出や得られた反例に基づいた修正効率についての定量的な評価や，抽象化の改善などが挙げられる．さらに，検証コストの削減のために，提案するモデル化手法に基づいてブループリントからモデルを自動で生成し，検証結果を元にブループリント上の修正箇所を特定するための手法の開発を目指す．

参考文献

[1] Ben-Ari, M., Pnueli, A. and Manna, Z.: The Temporal Logic of Branching Time, *Acta Informatica*, Vol. 20, No. 3, pp. 207–226 (1983).
[2] Cimatti, A., Clarke, E. M., Giunchiglia, F. and Roveri, M.: NuSMV: A New Symbolic Model Verifier, *Proc. 11th Int'l Conf. on Computer Aided Verification (CAV 1999) , LNCS1633*, pp. 495–499 (1999).
[3] Clarke, E. M., Grumberg, O. and Peled, D.: *Model checking*, MIT press (1999).
[4] EPICGames: Unreal Engine 4 Blueprint, https://docs.unrealengine.com/en-US/Engine/Blueprints/.
[5] Moreno-Ger, P., Fuentes-Fernández, R., Sierra-Rodríguez, J. L. and Fernández-Manjón, B.: Model-Checking for Adventure Videogames, *Information and Software Technology*, Vol. 51, No. 3, pp. 564–580 (2009).
[6] Pnueli, A.: A temporal logic of concurrent programs, *Theor, Comput, Sci,*, Vol. 13, pp. 45–60 (1981).
[7] Radomski, S. and Neubacher, T.: Formal Verification of Selected Game-Logic Specifications, *Proc. the 2nd EICS Workshop on Engineering Interactive Computer Systems with SCXML*, pp. 30–34 (2015).
[8] Rezin, R., Afanasyev, I., Mazzara, M. and Rivera, V.: Model Checking in Multiplayer Games Development, *2018 IEEE 32nd Int'l Conf. on Advanced Information Networking and Applications (AINA)*, pp. 826–833 (2018).
[9] 荒川巧也：Unreal Engine 4 ゲーム開発入門 第 2 版，株式会社翔泳社 (2019).
[10] 湊　和久：Unreal Engine 4 で極めるゲーム開発，株式会社ボーンデジタル (2019).