大規模RPGのためのシナリオ記述言語と統合開発環境の開発

Scenario Description Language and Integrated Development Environment for Large Scale Role Playing Games

清木　昌 [1)]
Masashi SEIKI

1) 東京大学大学院情報理工学系研究科コンピュータ科学専攻
（〒113-0033 東京都文京区本郷７丁目３番１号　理学部７号館　E-mail: mass@is.s.u-tokyo.ac.jp ）

ABSTRACT. In this project, we designed a scenario description language for video games that are categorized as RPG (Role Playing Games) or Adventure Games, and developed a prototype IDE (Integrated Development Environment) for the language. Target scenarios are developed with the IDE and finally transformed into XML data, which can be saved as files. However, scenario data are kind of *programs* whose executions are characterized by the state transition model. Scenarios developed with the IDE are more structured than scenarios written in traditional script languages for games. Hence, the language and the IDE improve productivity and enable us to check consistency between states in a scenario automatically.

1　背景

ゲーム業界は 2001 年出荷実績で国内約 4800 億円、輸出約 9700 億円 [1] と無視できない大きさの市場を持っており、ソフトウェア産業の中で輸出が輸入を大きく上回っている数少ない国際競争力が高い分野である。その中でもロールプレイングゲーム（RPG）と呼ばれる仮想世界の中で主人公とともに物語を疑似体験していく種類のゲームは幅広い年齢層に受け入れられ、全世界でシリーズ総計 4000 万本も出ているタイトルも生まれている。

近年は、三次元グラフィックスによるリアルな描写や DVD-ROM によるデータの大容量化を代表とするハードウェア技術の進歩により、ゲームでの表現の幅が広がっている。しかし、現実にはハードウェアの進歩にソフトウェアがなかなかついていけないという問題がある。この問題は、主に 3D 処理に起因するプログラムの複雑化や 3D モデルとそのモーションの作成によるコスト増大なども引き起こしているが、ここではそれには触れず、シナリオデータに着目して議論を進めたい。

記憶媒体の大容量化によって広く複雑な仮想世界を実現することが可能となった。これ自体は表現の可能性が広がるという点で喜ぶべきことである。しかし実際には、大きな容量がある以上それだけのものを作らないといけない、という本末転倒な状況が生まれている。大容量化によって実現が可能になった広く複雑な仮想世界。それを埋めるための大量のデータの作成にゲーム開発者は汲々となっている。実際、近年の RPG やアドベンチャーゲーム（ADV）のシナリオのテキスト量は俗に文庫数十冊分と言われる分量にふくれあがっているのである。

シナリオと言っても、映画のようなシーケンシャルな台本が１冊あればいいというわけではない。ゲームというものの価値の一部はその複雑性に依存しており、ユーザの選択によって様々に分岐し、因果関係が複雑に絡みあうようなシナリオが必要とされている。

例えば、ゲーム内では、鍵のかかっていない扉は押されれば開かないといけないし、通行人は話しかけられればその時の状況に応じた返答を返さねばならない。これがその場限りで他と独立しているものであればルールを記述するのは簡単だが、実際はそれほど単純ではない。多数のスイッチを押した順番によって罠を解除することが可能になるとか、さまざまな場所を訪れてフラグを立てておいたことによりクライマックスで正しい選択肢が現れるなどといった、複雑な因果関係と内部状態が存在するのである。

ここで、データを管理し、そのストーリー上の意味を定義するためのシナリオ記述言語が問題となる。それでは、シナリオ記述言語に必要な機能とは何だろうか。

最低でも、内部状態の保持、状態の更新、状態を参照しての条件分岐、などの機能が必要であるのは明らかである。しかし、先ほどから述べているデータの大規模化という文脈において、それだけの機能しか持たないスクリプト言語では究極の “スパゲッティプログラム” が生まれることになってしまう。それを避けるためには何らかの構造化を行うフレームワークが必要になるが、自由気ままに物語が遷移していくゲームシナリオにおいて、単純に通常のプログラムに用いる構造化の枠組みを適用しようとしても例外が多く発生するため難しい。

その上、この種のゲームで実際に用いられているシナリオ記述言語を調べてみると、ここで懸念しているような `if` や `goto` の機能しかもたないスクリプト言語も数多いという実情がある。従来のものの多くは BASIC や C 言語を非常に制限した形でコピーしたものであったり、ものによってはアセンブリ言語相当の条件分岐機能しか備えていないようなものなことさえある。内部状態を保存しておくための変数機能も、すべてグローバルでかつすべての変数を番号で管理しているというようなケースもごく普通のこととして存在している。これは各ゲーム会社が自社オリジナルのゲーム用スクリプトエンジンを開発・使用していく中で、厳しい納期の連続によりスクリプト言語を改善していけるだけの余力がプログラマに無かったということが大きな要因なのであろう。

```
⋮

*label1501
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
SHOWBG "background_01.gif"
SHOWIMG "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 番フラグによって実行されるイベント
SHOWBG "background_02.gif"
SHOWIMG "character_01_02.gif"
PRINT "メッセージ出力３"
WAIT

⋮

*label1504
; 一般の処理

⋮
```

図 1: 従来のシナリオスクリプトのイメージ

このようなスクリプト言語を使用することの代償として、大規模なシナリオを記述しようとした場合に大きな無理が生じるのは当然の帰結である。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/管理/デバッグすることを考えてみればよくわかるであろう。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り、納品直前に大量のシナリオ的なバグ[*1]が出ることになる。

現状では多くのデバッグ人員を雇い、多大なマンパワーをかけて人海戦術でバグ出しすることによって対応しているが、仮に選択肢を総当り検査したとして、その手数は選択肢数に対して指数関数的に増加するため、いつまでも続けられることではないであろう。

さらに、機能の十分でないスクリプト言語では対応が難しいのがネットワークゲームの分野である。各家庭への常時接続回線が本格的な普及を始めた今後、大きく伸びて来るであろうネットワークゲームにおいては、複数のプレイヤーが同時にゲーム世界に対して行動を起こした時の対応を、矛盾無く正確に記述できなければならない。ライブ性を持たせるために十分なテスト期間なしで頻繁にシナリオデータを変更・更新していかないといけないという事情もあり、従来の簡易言語では対応が困難であろう。

2 目的

このような現状を踏まえ、本プロジェクトは、一般にロールプレイングゲーム（RPG）やアドベンチャーゲーム（ADV）と呼ばれる、ユーザの入力に対して動的に変化するストーリーを主たる特徴とするゲームの開発において、そのシナリオデータの記述・管理・デバッグという観点から、開発の生産性を向上させる手法を模索することを目的とし、そのために、以下のことを行った。

1) シナリオの新しい実行モデルの提案
2) サポート機能を充実させた開発環境の整備
3) 自動検証機能の実装

これにより、各社各ゲームで独自に開発されているシナリオエンジンを統合できるという可能性を示し、その統一されたエンジン用にきちんと作りこまれた統合開発環境によって生産性を向上させるというアプローチを提案する。同時に、自動検証という手段により、デバッグコストが大きく削減できる可能性があることを示す。

3 方針

今回のプロジェクトを実施するに当たって、特に留意した点は「実用になるものを目指す」ということである。開発の現場で実際に問題になっているものへのアプローチであるので、理論だけで終わってしまうことはできない。そこで、実用性を満たすためには必要であるとして設計時に考慮した事項がある。本論に先立って、その事項について以下に述べる。

（1）柔軟な処理ができること

ゲームはその性質上、変則的な処理を数多く含む。すべてが単純なフレームワーク内で処理できるような内容であると単調になってしまうからである。例えば、ある条件下で強制的に割り込んで発生するイベントなどは通常の操作サイクルからみれば変則的と呼べるであろうし、ミニゲームの実行のために一時的に別のプログラムに実行を移して、その結果を取り込まないといけないことも多い。

そのため、実行モデルの理論的な美しさを優先するあまりに、柔軟性が損なわれることがないように配慮しなければならない。特に `if` と `goto` では容易に表現できる内容さえも記述できなくなることは絶対に避けないといけないことである。この条件を満たさなければ、利用者より「使えない代物」というレッテルを貼られてしまうことになるであろう。

（2）改造が容易なこと

ビジネス向けアプリケーションであれば、操作形態が標準的なアプリケーションの基準に沿っていればいるほど操作性があがり望ましい。しかし、アミューズメントソフトウェアにおいては見た目の違いというものが各ゲームの一番わかりやすいオリジナリティとなる。そのため、汎用のシナリオ実行システムとはいえ、ユーザインタフェイスや操作体系などは各ゲームごとに独自のものとできるよう、入出力部の改造が容易であることが望まれる。

（3）わかりやすいこと

これはゲーム用の開発システムだけの話ではないが、利用者の習得が容易であることは非常に重要な要件となる。特に、本件ではプログラマとして訓練されていない者でも容易に利用可能であることが求められる。

かつてはシナリオスクリプトの記述もプログラマ寄りの仕事であり、プログラマないしはその素養をもった者が入力を担当していた。しかし、近年のシナリオの大規模化によって、シナリオスクリプトを記述する人間がプログラマとして訓練されていない場合が多くなってきている。また、通常の作業手順ではシナリオライターがテキストとしてシナリオを書き起こし、それをスクリプト担当者がシナリオスクリプトの形に編集していくことになるのである

[*1] ストーリーの不整合、あるシーンから抜け出せない、エンディングまで到達できない、など

が、細かな演出などの指定をするために最初からシナリオライターがシナリオスクリプト自体を入力したいという要望も存在する。

このようにプログラマとして訓練されていない者でも容易に習得可能であろうとすると、テキストベースでキーワードを埋め込んでいく、いわゆる「プログラミング言語」というアプローチでは限界が存在する。プログラミング初心者にもすぐに使えるようなシナリオの実行制御用言語ということで考えると、ラベルと goto 文による直感的にわかりやすい制御構造以上のものを望むことは難しい。しかし、それでは前に述べたシナリオスクリプトの管理の困難さが付きまとう。これを解決するには、テキストベースのアプローチから転換することが必要であろう。

4 手法

最大の目的はゲーム開発の生産性の向上であり、そのためにシナリオ管理の側面から改善を試みる、という流れはすでに述べたとおりである。シナリオ管理の生産性を上げるということは、すなわちシナリオデータの編集ソフトウェアの使いやすさを向上させることに他ならない。従来は、編集ソフトウェアはテキストエディタであり、編集ソフトウェアと独立した独立したデータ変換ソフトウェアによってシナリオスクリプトからゲーム実行時に読み込まれるシナリオデータを作成するというケースが多かった。しかし、先に述べたとおりテキストエディタでは生産性の向上に限界が存在する。そこで、シナリオデータを専用に編集可能で、データ変換も統合的に実行可能な編集ソフトウェアを使用することを前提として議論を進めていく。以降、この統合的な機能を持つシナリオデータの編集ソフトウェアを開発環境と呼ぶことにする。

本プロジェクトでは、開発環境を使いやすくするにあたって次に述べる手法を提案し、実行した。なお、時間的な制限もあり、下で述べるすべての機能は実装できていないことをあらかじめ断っておく。実装の詳細は後の章の説明を参照されたい。

（1）シナリオの新しい実行モデルの提案

この論文では状態遷移モデルに基づいたシナリオの実行モデルを提案する。すなわち、シナリオの各シーンをひとつの状態だとみなし、ストーリーの進行をシーン間の状態遷移とみなすのである。

従来のシナリオスクリプトは、スクリプトに記述されたものを上から順番に実行するという手続き型のアプローチをとっていた。実行モデルとしては理解しやすく、遷移関係を一度に把握できる規模のシナリオであれば十分に効果的でもある。また、スクリプトファイルを章や場所ごとに分割して管理するという方法で１ファイルあたりの分量を抑えている。

しかしながら、それでは局所的な遷移関係は十分に管理できたとしても、大規模なシナリオにおいて大局的に影響を及ぼしあうフラグが存在した場合に動作を追跡することが困難である。また、同じシーンが異なる章・異なる場所を通して存在する場合に、それぞれのスクリプトファイルへ大量にコピー&ペーストしなければならないというケースも多い。

これらはテキストエディタで編集を行うスクリプト言語であるがゆえに、見やすさを優先すると１ファイルに納めないといけないことからくる制約である。開発環境を新たに整えることを考えたときに、この制約にとらわれる必要はない。そこで、シナリオをシーンの集合とみなし、開発環境はシーンの集合に対して操作を行うというモデルを考える。今後、状態遷移を考える際の単位となるシーンをブロックと呼ぶ。

このモデルを取り入れると、開発環境に操作対象のシーンを場所や時間といった要素で検索・絞り込む機能を持たせることにより、従来のファイルごとの管理よりももっと柔軟に各シーンの関係を見ることが可能になる。また、各ブロックごとの遷移関係を自動的に解析することによって、あるブロックに飛んでくる元はどこがあるのかということや、そのブロックの次に実行されるブロックがどこなのかなどを、シームレスに表示することも可能である。

そして、シナリオブロック間の状態遷移というこのモデルは、後に説明する自動検証機能を綺麗に実現させることにも貢献している。

（2）サポート機能を充実させた開発環境の整備

シナリオをシーンの集合と見なすと、開発環境に様々な機能を持たせることが可能になる。前節で述べたようなシーンの絞り込み機能や前後関係の解析、そして次節で述べる自動検証機能などである。

開発環境においては GUI によるオペレーションを基本とすることにより、プログラミングに熟練していない使用者でも直感的な操作を可能とする。最終的には、シナリオ文面の入力以外はフル GUI オペレーションができるようにすることを目指す。

また、開発環境は裏にある実行モデルを包み隠すという重要な役割もある。もしも状態遷移モデルのスクリプト言語をテキストファイル上で行おうとすると、遷移関係をすべてのシーンについて書かないといけないという悪夢のような事態になるが、専用のソフトウェアで包み隠して自動的に処理することによって、利用者が実行モデルの詳細を意識する必要をなすことができる。

すなわち、最終的な形態として、利用者は GUI 上で関連するシーンをグルーピングし、実行順に並べるだけで、遷移関係を指定でき、分岐も直感的なイメージに近いフローチャートのような絵で書けるようなインターフェイスを目指す。

また、実行モデルとは関係がないが、変数名の補完機能やコメントの見やすい表示の機能なども統合開発環境ならではの生産性向上のための機能であろう。

（3）自動検証機能の実装

生産性向上のために最も貢献すると思われるのが自動検証機能である。さまざまなシナリオ上の性質をワンクリックで検証する。

検証できる性質の例を挙げてみると

実行不能ブロックの存在 初期状態からどうやっても到達することのできないブロックを発見する。何らかのバグであることが想定できる。

終了不能状態の存在 そこからどんな遷移をしたとしてもエンディングまでたどり着けない状態を発見する。俗にハマり状態と呼ばれており、絶対に存在してはならない状態である。

各種 invariant あるフラグとあるフラグは同時に on になることはない、などといった性質がシナリオ上で常に成り立っているかを検証する。

これまで、多くのマンパワーを費やして総当りチェックに近い形でシナリオのデバッグは行われていたが、ある種の致命的なバグに関しては、この自動検証機能によって完全に存在しないことをマンパワーをかけずに保証できるようになる。

5 システム構成

本プロジェクトで作成したシステムの構成を図 2 に示す。シナリオの実行モデルとそのデータの構造を設計したのちに、それを実行するエンジン部分を作成し、同時に

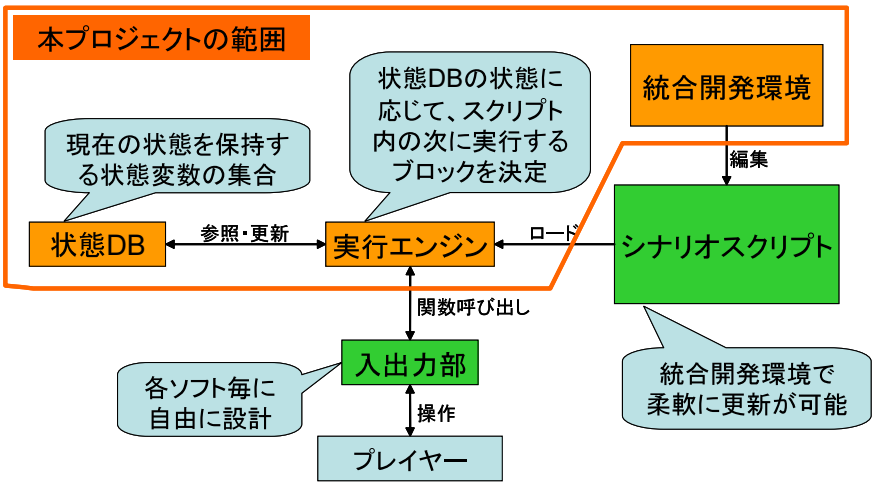

図 2: システム構成

データを作成する開発環境を作成した。

一つのポイントとして、入出力部は独立しているということが挙げられる。3 章で述べた通り、入出力部は各作品毎に独自性を出さなければならない部分であり、スクリプト実行エンジンとは切り離して自由に作成できるようにしなければならない。現状の実装では、入出力用のインターフェイスを定め、それを実装するクラスとして入出力部を作成するようになっているが、本来は初期化関数だけ規格化し、初期化関数内で各機能に対してコールバック関数を登録するようなプラグイン方式にするべきであろうと考えている。どちらにせよ、シナリオの実行制御と、実際の画面表示やユーザからの入力の受付などの部分は切り離して実装し、実行制御部分を各ゲームで共通化させることは可能である。

6 状態遷移に基づくシナリオ実行モデル

（1）状態遷移モデルの実装

状態遷移に基づくシナリオ実行モデルのイメージはすでに述べたとおりであるが、実際に各ブロックを入力する方法を考えたときに工夫が必要となる。すべての遷移関係を指定するのは煩雑であるし、管理上の要請でブロックをグループに分割管理する必要が出てきた際に完全にフラットなブロック群があるだけではやりにくい。

そこで、ゲーム内のフラグや実行状態などを保持する状態変数に着目し、個々の状態は状態変数の値によって定義することにする。そして、状態変数の値に関する条件を各ブロックの実行条件とすることとした。すなわち、状態変数 x_0〜x_n が存在し、それぞれの値域を V_i としたときに、各ブロック B_j の実行条件 C_j は $C_j \subset \prod V_i$ と表現することができ、$(x_0, \ldots, x_n) \in C_j$ が成立する時に B_j は実行される。このとき、基本的に $\forall j \forall k\ C_j \cup C_k = \phi$ とする。

そして、各ブロックは基本的に各状態変数 x_i に関する条件でグルーピングされることになる。具体的には、ツリー状に各ブロックは構造化され、そのツリーのノードには状態変数 x_i が振られ、そのノードから子ノードへのパスに x_i の値に関する条件が割り当てられる形になる。各ブロックの実行条件は、ルートからそのブロックに至るまでのパスで定められた各状態変数への条件の論理積をとったものになる。

もちろん、これだけでは柔軟性に欠け、また入力も非常に煩雑になる。最終的には、GUI による簡略入力により、よく出てくる実行パターンに関しては自動的に状態変数が割り当てられ、遷移関係も自動で補完されるようにしなければならないが、まだ実装されていない。

（2）実行アルゴリズム

1) 状態変数 x_i を初期化する
2) if $\exists j\ s.t.(x_0, \ldots, x_n) \in C_j$
   then goto 3, else raise error
3) ブロック B_j のスクリプトを実行（スクリプト内で状態変数の書き換えが行われる）
4) ブロックを実行した結果、終了フラグが立っていれば実行終了
5) 2 に戻る

（3）状態変数

現在の実装では、状態変数は型を持ち、型にはさらに型のクラスが存在している。現状では変数の型のクラスには列挙型を表す Enumerate クラスしか存在しない。Enumerate クラスを実際に取る値を与えることでインスタンシエイトしたものが、各状態変数の最終的な型となる。

（4）シナリオデータの記録形式

内部でツリー状に保管されているシナリオデータは XML 形式で出力される。シナリオデータには、状態変数の型に関する情報と、シナリオブロックの集合が含まれる。以下の単純なスクリプトは false と true という値を持つ Enumerate 型クラスから生成された Flag 型をまず定義し、その後 Flag 型で初期値が false な変数 Flag1 を定義している。所々にある Text エレメントは Print 命令の中以外は全てコメントである。コメントは開発環境内で確認・編集が可能となっている。

```
<?xml version="1.0" encoding="utf-8"?>
<LapisData>
  <Definition>
    <VariableClasses>
      <Enumerate>
        <Name>Flag</Name>
        <Items>
          <Item>
            <Value>false</Value>
            <Text>偽</Text>
          </Item>
          <Item>
            <Value>true</Value>
            <Text>真</Text>
          </Item>
        </Items>
      </Enumerate>
    </VariableClasses>
    <Variables>
      <Variable>
        <Text>フラグの一つ</Text>
        <Name>Flag1</Name>
        <Class>Flag</Class>
```

```
        <InitialValue>false</InitialValue>
      </Variable>
    </Variables>
  </Definition>
  <Scenario ID="Root##1">
    <Switch ID="Switch##1">
      <Text>変数値による振り分けのサンプル</Text>
      <Variable>Flag1</Variable>
      <Cases>
        <Case Condition="false">
          <Block ID="Block##1">
            <Let ID="Let##1">
              <Text>代入のサンプル</Text>
              <Variable>Flag1</Variable>
              <Value>true</Value>
            </Let>
            <Print ID="Print##1">
              <Text>文章を出力して、終了します。</Text>
            </Print>
          </Block>
        </Case>
        <Case Condition="true">
          <Block ID="Block##2">
            <End ID="End##1" />
          </Block>
        </Case>
      </Cases>
    </Switch>
  </Scenario>
</LapisData>
```

図 3: XML 形式で出力されたスクリプトの例

XML で出力することにより、XSLT などの変換ソフトウェアで任意の部分を取り出して任意の形式に変換できる。もっとも単純な例では、Print 命令内の Text エレメントの内容だけを取り出すことによって、シナリオ文面の確認用のテキストファイルを作成することも可能である。

7 開発環境の整備

開発環境は Microsoft .NET Framework 上の C♯ 言語を用いて開発した。利用者の層から考えて、開発環境は Windows 上で動かなければならないという点、そして、できるだけ見栄えのいい UI が利用可能かという点を考慮した上で、短期間で開発が可能な高生産性の言語という理由で C♯ を採用した。

GUI による変数やブロック管理、そして任意のブロック実行条件による編集対象ブロックの絞り込みなどを実装している。また、開発環境のメニューからシナリオを実際に実行することができ、次章で説明する自動検証機能も実装した。

しかし、あるブロックの遷移関係の前後をたどる機能は未実装である。自動検証の中で遷移関係の情報は集めてるので、表示方法さえ決めてしまえば実装は容易であると考えている。

また、本来は入出力部はこのプロジェクトの範囲外ではあるが、サンプルの実行のためにデジタルノベル風の入出力部を作成し、組み込んでいる。

8 自動検証機能

自動検証の技術は主に電気回路や通信プロトコルの分野で活用されてきた [2]。明示的に各状態を数え上げていた当初は、計算機の性能の問題もあり数百から数千の状態数のものしか検証できなかったため、規模の小さい回路やプロトコルの検証に利用されていた。しかし、BDD (Binary Decision Diagrams)[3] を用いた記号モデル検査 [4] や抽象モデル検査などの理論の発展により、10^{1300} もの状態数を持つ回路に対してもいくつかの性質を検証できるようになっている [5]。

しかし、この技術をゲームのシナリオに適用しようとした例はほとんどない。1990 年に多人数で遊ぶゲームブックに対して通信プロトコルの検証ツールを適用しようとした論文 [6] が残っているが、それもその後の展開はなかったようである。

しかしながら、ゲームのシナリオを状態遷移系と見なした際に、自動検証の手法は非常に有効となりえる。発売後に修正が効きにくいコンシュマーゲーム機用のゲームソフトにとって、エンディングに到達できないなどのバグは時に製品回収にまでつながることがある。これらの致命的な障害を確実に発見できる手段を提供することは意義のあることである。

現状では、以下の 2 つの性質が検証可能になっている。

（1）到達不能ブロック

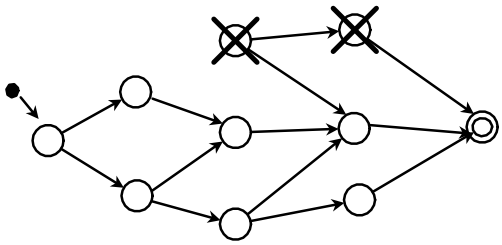

図 4: 到達不能ブロック

図 4 の状態遷移図に×印で示されているようなブロックを発見する。なお、正確にはブロック ≠ 状態であるので、到達可能な状態変数値の組み合わせでは一度もアクティブにならないブロックを発見する、という表現が正しい。

現在の実装では、初期状態から到達可能な変数の組み合わせを explicit に数え上げるアルゴリズムを採用している。サンプルで用いた 60 状態 30 ブロック程度のシナリオデータであれば、次の終了不能状態チェックと併せてほぼ瞬時に検証を完了させているが、規模を大きくした場合には大きな性能低下が予想される。

（2）終了不能状態

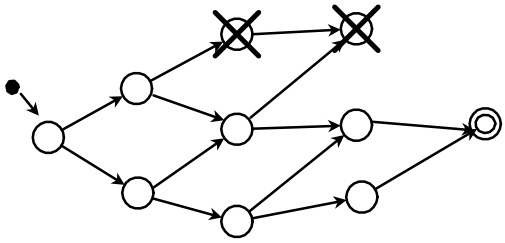

図 5: 終了不能状態

図 5 の状態遷移図で×印となっているような状態を発見する。

アルゴリズムとしては、到達不能ブロックをチェックする際に作成した到達しうる状態（変数値の組み合わせ）とその間の遷移情報を用い、遷移関係を逆向きにして終了状態からたどっていくことにより、終了状態に到達できない状態を発見する。ここで見つかった終了不能状態は、状態を表す変数値の組み合わせの形で報告される。

9 評価と今後の課題

（1）開発環境の UI の充実

開発環境の GUI 機能の充実は生産性の向上にとってとても重要な要素であるが、時間的な制約により、現状では十分に使いやすいといえるものにはなっていない。これが今後の課題となる。

特に、現状ではシーケンシャルなシーンの連続があったときにでも、いちいち状態変数を割り当て、その変数への代入という形でシーンを遷移させないといけない仕様になっている。こういった典型的なパターンについては GUI による省略入力法を用意しておき、利用者は直感的な最低限度の操作で実現できるようにしなければならない。

（2）自動検証機能の強化

現在は explicit な数え上げを行っているが、状態数が多くなると計算時間が非実用的なレベルになってしまいかねない。BDD を使った記号モデル検査の手法などを用いてできるだけ時間・空間計算量を減らしていかねばならない。

十分に効率化した上では、最終的に有限の結末に収束するというゲームシナリオの性質により、本質的に問題となる状態数はそれほど多くならないと考えられるため、シナリオサイズが大きくなっても実用的な速度で検証可能であろうと考えている。

また、自動検証の内容も、任意の invariant を指定できるように改良を加える。最低限、CTL (Computation Tree Logic) 程度の時相論理式は検証できるようにする予定である。

（3）もっと実用になるための様々な改良

実際の開発現場では、テキストベースのスクリプトファイルというのはさまざまな形で便利に使われている。テキストファイルに多少の予約語が入っただけのファイルであるので、少し編集してセリフの音声収録用の台本としたり、スクリプトファイルをそのまま印刷して誤字チェックを行ったりと、テキストファイルであることの柔軟性を発揮していることがしばしばある。

そういった手軽にシナリオ文面を加工したいという需要にも応えられるように、開発環境からシナリオデータを様々な形式でエクスポートできることが望ましい。特定のキャラクターのセリフのみを抽出してテキストファイル化することなどが簡単に行える機能などは需要があるだろう。開発のさまざまな工程の補助ができてこその統合開発環境である。

また、今回はシナリオエンジンを自前で持つことを前提としたプロジェクトであったが、シナリオエンジンそのものは既存のものを流用し、編集だけを開発環境で行うというアプローチもある。元のシナリオデータとしては状態遷移モデルで１シーン１ブロックという形で保持し、その上でデータ入力と管理も行うが、実行する際には開発環境の変換出力機能によって従来の `if, goto` のスクリプト言語にコンバートするという手法である。この方法でもこれまで述べてきたような生産性向上の効果を十分に上げることが可能であり、なおかつ従来のプログラム資産が流用可能となる。そうすれば、適用範囲が飛躍的に広がることが期待できる。

（4）ネットワークゲームへの対応

最終的には、ネットワークゲームにおける並行して複数のキャラクターがアクションを起こすモデルにも対応したい。並行実行が起こると、検証の手間は跳ね上がり、人力でチェックを行うことが非現実的となる。そういった部分でこそ自動検証の力が真価を発揮することになると考えている。

10 まとめ

今回のプロジェクトでは、ゲームの開発現場における生産性の向上ということをテーマに、新しいシナリオ実行モデルを提案し、実際にその開発環境のプロトタイプを作成した。そして、開発環境に自動検証の機能を付加した。自動検証をゲームのシナリオに適用するというアプローチは他に例を見ない。

残念ながら、生産性の向上を実感するには開発環境の作り込みがまだ足りないが、ワンクリックで論理的なバグを発見してくれる自動検証の能力は、従来の人も時間もかかるデバッグの姿を変革できる可能性を感じさせるものであった。

今後、現在以上にコンピュータ技術が発展するに従って、ゲーム上の仮想世界はますます複雑で高度になっていくであろう。今回提案した、きちんと整備されている統一された開発環境と、その上で動く自動検証機能というアプローチが、そんな新しい楽しみの創造の流れを支えていく基盤技術として活躍できると確信している。

11 参加企業及び機関

プロジェクト管理組織として株式会社メディアフロントに事務処理を中心にご協力いただきました。

謝辞

本プロジェクトの遂行に当たって、プロジェクトマネージャであり、またこの題材の研究の偉大なる先輩でもある電気通信大学の竹内郁雄教授より、多くのアドバイスと励ましをいただきました。また、慣れない事務処理に戸惑う中で株式会社メディアフロントの小松様、大江様、前野様、岡本様をはじめとした皆様方に多大なるサポートをいただきました。あわせて、未踏ソフトウェア創造事業という先進的な事業を維持されている IPA の関係者各位、さまざまな機会にご意見をくださいました皆様、そして未踏ユースへ注力する自分を暖かく見守ってくれた研究室の諸氏に深く感謝の意を表します。

参考文献

[1] CESA：2002 CESA ゲーム白書，p192，社団法人コンピュータエンターテインメント協会，東京 (2002)

[2] Edmund M. Clarke and Orna Grumberg and Dron A. Peled：Model Checking，p330，The MIT Press，Cumberland(1999)

[3] Randal E. Bryant：Graph-Based Algorithms for Boolean Function Manipulation，ieeetc，Vol.C-35,NO8，p677〜691(1986)

[4] J.R. Burch and E.M. Clarke and K.L. McMillan and D.L. Dill and L.J. Hwang：Symbolic Model Checking: 10^{20} States and Beyond，Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science”，p1〜33，IEEE Computer Society Press，Washington, D.C.(1990)

[5] Edmund M. Clarke and Orna Grumberg and David E. Long：Model Checking and Abstraction，ACM Transactions on Programming Languages and Systems，Vol.16,NO5，p1512〜1542(1994)

[6] 柴崎雅史：多人数ゲームシナリオの記述と検証法，情報処理学会研究報告 マルチメディア通信と分散処理，Vol.47,NO5(1990)