Automatic verification for node-based visual script notation using model checking

Isamu HASEGAWA[1] and Tomoyuki YOKOGAWA[2]

[1] SQUARE ENIX CO., LTD. haseisam@square-enix.com
[2] Okayama Prefectural University t-yokoga@cse.oka-pu.ac.jp

Abstract. Visual script languages with a node-based interface have commonly been used in the video game industry. We examined the bug database obtained in the development of FINAL FANTASY XV (FFXV), and noticed that several types of bugs were caused by simple mis-descriptions of visual scripts and could therefore be mechanically detected.
We propose a method for the automatic verification of visual scripts in order to improve productivity of video game development. Our method can automatically detect those bugs by using symbolic model checking. We show a translation algorithm which can automatically convert a visual script to an input model for NuSMV that is an implementation of symbolic model checking.
For a preliminary evaluation, we applied our method to visual scripts used in the production for FFXV. The evaluation results demonstrate that our method can detect bugs of scripts and works well in a reasonable time.

Keywords: Formal methods Symbolic model checking Visual script Game development

1 Introduction

In the recent video game industry, game designers write game logic using script languages. Since most of game designers are not familiar with writing programs, the use of visual script languages allow designers to perform such scripting operation, and thus help improve the productivity of game logic development. In particular, visual script languages with a node-based interface are widely used in game development.

However, it is hard to maintain game logic written in visual script languages because they can quickly become large and complicated during the course of production, and thus become hard to verify or modify, and very prone to human error.

We examined the bug database obtained in the development of FINAL FANTASY XV (FFXV) [7], and noticed that several types of bugs were caused indeed by simple mis-descriptions of visual scripts. A system that can automatically detect such mis-descriptions would had been a great help to our production.

Since most visual script implementations could be treated as a kind of state machine [8], and model checking is a well-researched technique to automatically verify finite state machines [5], we propose in this paper a method for automatic verification of visual script notations with symbolic model checking [3] for efficient game production. Our main contributions are the following. (1) To apply symbolic model checking to verify visual scripts, we provide a translation algorithm from a visual script description to an input model for NuSMV [5], that is an implementation of symbolic model checking. (2) We show a preliminary evaluation of our method by applying it to visual scripts which are produced in the development of FFXV, and demonstrate that most of the verification tasks are completed in a realistic amount of time.

The rest of this paper is organized as follows. We first introduce prerequisite topics and show a motivating example in section 2. Section 3 explains the proposed method. Section 4 provides the translation algorithm from a visual script to an input model which can be accepted to NuSMV. Section 5 explains how to write node semantics. We show the results of our preliminary evaluation in section 6 and conclude our work in section 7.

2 Background

2.1 Model Checking

Model checking is an automatic technique for verifying correctness properties of a finite-state system [6]. The verification procedure is performed by an exhaustive search over the state space. Since the size of the state space exponentially increases with the number of system components, it is difficult to apply model checking to large-scale systems. Symbolic model checking can efficiently handle large-scale systems by replacing explicit state representation with boolean formula.

NuSMV [5] is one of the most successful implementations of symbolic model checking. The model verified by NuSMV is written by a specific input language (called SMV language). The properties to be checked is expressed by temporal logic LTL (Linear Temporal Logic) [14] and CTL (Computational Tree Logic) [1].

```
MODULE main
VAR
  sw : {on, off};
ASSIGN
  init(sw) := {on, off};
  next(sw) := case
    sw = on : off;
    TRUE : sw;
  esac;
CTLSPEC AG (AF sw = on)
```

Fig. 1. An example model described in SMV language

Fig. 1 is an example of an input model to NuSMV. The input model described by SMV language is composed of variable declaration part (described by `VAR`) and transition relation definition part (described by `ASSIGN`). The property is expressed as a LTL formula (described by `LTLSPEC`) or a CTL formula (described by `CTLSPEC`).

This example has one variable `sw` which may have one of the two values `on` and `off`. In its initial state, either `on` or `off` is assigned to `sw` non-deterministically. In the case that `sw` is `on`, `sw` becomes `off` in the next state, or `sw` does not change its value. Thus the sequence of the value of `sw` can be either `on`, `off`, `off` ... (when the initial value is `on`) or `off`, `off` ... (when the initial value is `off`). The CTL formula in this example has two CTL operators **AG** and **AF**. **AG** represents "in Any path" and "Globally," and **AF** represents "in Any path" and "in the Future." This formula expresses the following property: the system always satisfies that `sw` necessarily becomes `on`. When the model is inputted to NuSMV, NuSMV returns `FALSE` for this property because there is a path where `sw` continues to be `off`. Fig. 2 shows the result and the counterexample generated by NuSMV. The counterexample shows the path where `sw` continues to be `off`.

```
-- specification AG (AF sw = on)  is false
-- as demonstrated by the following
   execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
sw = on
-- Loop starts here
-> State: 1.2 <-
sw = off
-> State: 1.3 <-
```

Fig. 2. A counterexample generated by NuSMV

2.2 Motivating Example

Many game development environments have their own visual scripting system such as the Blueprint in Unreal Engine [18], [12], [9]. Although there are slight differences among each of visual scripting system, their syntax and semantics are basically the same. In this paper, readers can assume the Blueprint [18] as the visual scripting system since its syntax and semantics are very similar to our in-house visual scripting system.

In the development with node-based visual script languages, logic is described as a *node graph* which is composed of *nodes* and *edges*. Nodes express values, variables, arithmetic operators, or control statements of the visual script, which correspond to statements in text-based script languages such as if/while-statements, assignments, and so on. Since the purpose of visual scripts is to control game components such as sound, visual effect, and so on, many nodes express invocations of APIs of those components. For example, "Play SE" node notifies sound component of the game system to start playing sound effect, "Fade Out" node notifies screen effect component to start fade-out effect [3]. Edges connect nodes through input and output *ports*, and express data and control flows.

Fig.3 shows an example of visual script. Note that we omitted data flow edges in fig. 3 such as the condition value inputted to If node. This is because our method does not address the detection of bugs caused by an illegal data flow. This example has the following behavior:

- When Movie Clip node receives an input signal through the Start port, it starts playing the movie clip, and sends output signal through the Finished port when it finishes playing. If the movie clip is skipped by a game player, it sends output signal through the Skipped port instead of the Finished port.
- The Set Event Mode node modifies the global flag variable "event mode". When it receives the input signal through the Enable or Disable port, the event mode flag becomes true or false respectively. This example includes two Set Event Mode nodes, and both of them modify the same variable instance since the "event mode" variable is not a variable in the script but a variable in the external game system.
- If node is used for conditional branch like if-statement in text-based languages. Its condition value is inputted through data flow port. As stated above, we omit such ports.
- The global flag variable "event mode" must be true during playing the movie clip, and must be false otherwise in order to change some game state during playing movie such as disabling gamepad, etc.

Note that Movie Clip node has its own state transition, and sends output Finished or Skipped independently from the original control flow. It means that there can be multiple activated nodes and multiple activated control signals in a graph. It is one of the significant differences of visual script languages from Statecharts and a reason that we can not directly apply prior research to visual script languages.

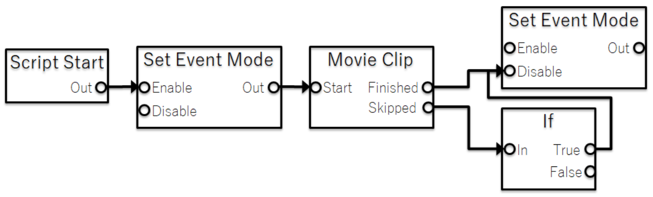

Fig. 3. An example of node-based visual script (including a typical bug)

[3] "fade out" is a gradual transition from the game screen to blank image, used in movies, games, etc.

This example contains a bug that actually often occurred during the development of FFXV. It appears that the False port of the If node has no connection. Therefore, if Movie Clip branches to Skipped and then If branches to False, the event mode flag is not changed and remains to be true. It causes incorrect behavior since the event mode flag is true after playing the movie.

There were a wide variety of similar bugs during the development of FFXV, e.g. "BGM is not changed correctly in some cases.", "Enemy characters never respawn in a specific condition.", and so on. Moreover, since many game logic scripts are written by game designers who are not familiar with writing programs, scripts often become large and complicated. Therefore, it is tough to find those bugs by visual inspection, even though these are caused by trivial misdescriptions such as missing one node, or missing one edge, and so on. Our goal is to detect those large amounts of trivial but hard-to-find bugs automatically and exhaustively. Since our products already have a lot of massive scripts, we should cover not only newly written scripts but also those existing scripts.

2.3 Related Work

Video games essentially have a large number of combinations of internal states and external stimuli. This makes it difficult to detect problems which come out under specific conditions by testing. Model checking has been applied to video game developments since it can solve such problems by exhaustive verification. Moreno-Ger et al. [13] proposed a method for verifying game scripts created in ⟨e-Adventure⟩ platform using NuSMV. Radomski et al. [15] showed a framework in which video game logics are modeled by State Chart XML (SCXML) formalism and their properties can be checked by the SPIN model checker. Rezin et al. [16] developed a method to model a multi-player game design as a Kripke structure and to verify it by NuSMV. These studies show that applying model checking to video game development is very promising and application to game logic described by node-based visual scripts is also expected.

There have been a number of studies that have applied model checking to verification of node-based state transition system designs. Statecharts and its variants, such as UML state machine [17] and RSML (Requirements State Machine Language) [11], are one of the most popular notations for describing state transition systems in a node-based manner. Chan et al. [4] provided a translation from RSML notation to a model described by SMV language. This translation procedure encodes components of the inputted RSML by SMV variables and expresses changes of the components as transition relation. Zhao et al. [19] studied representation of Statecharts step-semantics as a Kripke structure, which is a graph-based state transition representation, and carried out verification using SMV model-checker. Jussila et al. [10] presented a representation of a subset of UML state machines as Promela which is an input language of SPIN model-checker.

In the semantics of Statecharts and its variants, their nodes represent states and only simple actions (enter/exit or do action in the case of UML state machine) can be assigned to each node. While in the visual script notations that

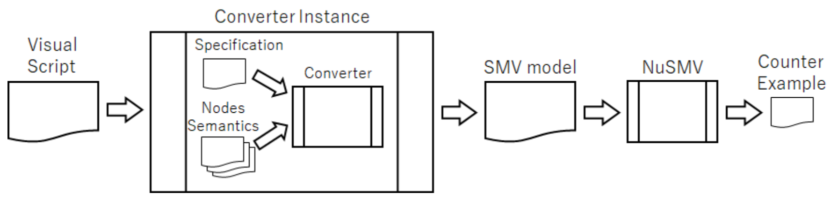

Fig. 4. System overview

we focus, each node expresses some game logic computation which can be performed individually and can have a particular semantics. Thus it is difficult to directly apply the existing procedures to the verification of such a visual script notation. In this paper, we propose a method to translate from visual scripts to models by SMV language.

3 Approach

3.1 System Overview

Fig. 4 shows the system overview of the visual script verification environment with NuSMV. This environment carries out verification by converting a visual script into an SMV model. First, the system generates a converter instance from specifications to be checked and the corresponding node semantics. Then the visual script is converted into an SMV model by using the converter instance. NuSMV can verify whether the inputted visual script satisfies the specifications or not. When the specifications are not satisfied, NuSMV outputs counterexamples.

In this section, we explain the overview of the SMV models that our method generates from visual scripts (section 3.2), specifications to be checked (section 3.3), and how to detect bugs using counterexamples (section 3.4).

3.2 Model Overview

We first show the overview of SMV model generated by the proposed method with an example. Fig. 5 is an SMV model converted from the visual script shown in fig. 3.

SMV Variables We prepare four types of SMV variables to describe the behavior of a visual script.

– *Input* and *output variables* represent activated ports of each node in visual script. Since only one input/output port can be activated at the same time

```
MODULE main
VAR
  ScriptStart1Out  : {none, Out};
  SetEventMode2In  : {none, Enable, Disable};                     -- (1)
  SetEventMode2Out : {none, Out};                                 -- (1)
  MovieClip3In     : {none, Start};
  MovieClip3Out    : {none, Finished, Skipped};
  MovieClip3State  : {Stopped, Playing, Finished, Skipped};       -- (2)
  SetEventMode4In  : {none, Enable, Disable};
  SetEventMode4Out : {none, Out};
  If5In            : {none, In};
  If5Out           : {none, True, False};
  EventMode        : {true, false};                               -- (3)
FAIRNESS MovieClip3State = Stopped;                               -- (4)
ASSIGN
  init(ScriptStart1Out) := Out;
  next(ScriptStart1Out) := none;
  init(SetEventMode2In) := none;
  next(SetEventMode2In) := case
    ScriptStart1Out = Out : Enable;                               -- (5)
    TRUE                  : none;
  esac;
  init(SetEventMode2Out) := none;                                 -- (6)
  next(SetEventMode2Out) := case                                  -- (6)
    SetEventMode2In = Enable | SetEventMode2In = Disable : Out;   -- (6)
    TRUE                                                 : none;  -- (6)
  esac;
  init(MovieClip3In) := none;
  next(MovieClip3In) := case
    SetEventMode2Out = Out : Start;
    TRUE                   : none;
  esac;
  init(MovieClip3Out) := none;
  next(MovieClip3Out) := case
    MovieClip3State = Finished : Finished;                        -- (7)
    MovieClip3State = Skipped  : Skipped;                         -- (7)
    TRUE                       : none;
  esac;
  init(MovieClip3State) := Stopped;                               -- (8)
  next(MovieClip3State) := case
    MovieClip3In = Start      : Playing;                          -- (9)
    MovieClip3State = Playing : {Playing, Finished, Skipped};     -- (10)
    TRUE                      : Stopped;                          -- (11)
  esac;
  init(SetEventMode4In) := none;                                  -- (12)
  next(SetEventMode4In) := case
    MovieClip3Out = Finished : Disable;                           -- (13)
    If5Out = True            : Disable;                           -- (13)
    TRUE                     : none;                              -- (14)
  esac;
  init(SetEventMode4Out) := none;                                 -- (15)
  next(SetEventMode4Out) := case                                  -- (15)
    SetEventMode4In = Enable | SetEventMode4In = Disable : Out;   -- (15)
    TRUE                                                 : none;  -- (15)
  esac;
  init(If5In) := none;
  next(If5In) := case
    MovieClip3Out = Skipped : In;
    TRUE                    : none;
  esac;
  init(If5Out) := none;
  next(If5Out) := case
    If5In = In : {True, False};                                   -- (16)
    TRUE       : none;
  esac;
  init(EventMode) := false;
  next(EventMode) := case
    SetEventMode2In = Enable | SetEventMode4In = Enable   : true; -- (17)
    SetEventMode2In = Disable | SetEventMode4In = Disable : false;-- (18)
    TRUE                      : EventMode;
  esac;

CTLSPEC AG(EventMode = true -> AF(EventMode = false))             -- (19)
```

Fig. 5. Converted SMV model

in most cases[4], we declare just one input/output variable for one node even if the node has two or more input/output ports. E.g. `SetEventMode2In` and `SetEventMode2Out` (fig. 5 (1)) are the *input* and *output* variable for the leftmost Set Event Mode node in fig. 3. The value domains of *input*/*output* variables are the names of ports and special value `none` which represents that no port is activated. E.g. since Set Event Mode node has 2 input ports `Enable` and `Disable`, the value domain of `SetEventMode2In` is `none`, `Enable`, and `Disable`. When the value of `SetEventMode2In` is `Enable`, it means that the input port Enable is active in the leftmost Set Event Mode node.

- *Script variables* represent variables used in visual scripts and states of external components that the visual scripts interact with. E.g. the global flag "event mode" stated in 2.2 is a flag variable of the external game system that the visual scripts interact with, and is declared as a *script variable* `EventMode` (fig. 5 (3)). The specification often specifies the correct behavior of those variables.
- *State variables* represent the internal state of each node whose semantics has state transition. E.g. `MovieClip3State` is the state variable for Movie Clip node (fig. 5 (2)).

Control Flow An edge in visual scripts express a portion of control flow that is defined as a set of output port O and input port I, where I is activated iff O is activated. Therefore, we can describe an edge as a value definition of an input variable according to values of output variables in SMV models. E.g. the value of `SetEventMode2In` becomes `Enable` when the value of `ScriptStart1Out` is `Out` (fig. 5 (5)). It describes that Out port of Script Start node is connected to Enable port of the leftmost Set Event Mode node.

Thus the value transitions of *input* and *output* variables express the control flow in visual scripts. For example, assuming that the control flow of Fig. 2.2 is: `ScriptStart:Out` → `SetEventMode:Enable` → `SetEventMode:Out` → `MovieClip:Start` → `MovieClip:Skipped` → `If:In` → `If:False` , and fig. 6 shows the value transitions in this case.

Node Semantics Value definition of *output variables*, *script variables* and *state variables* are specified by semantics of each node. E.g. the node semantics of Set Event Mode is: "when it received input signal Enable or Disable, it edits the global flag EventMode respectively, and immediately output signal through Out port." This node semantics corresponds to the definition of the variables `SetEventMode2Out` and `EventMode` (fig. 5 (6), (17)).

3.3 Specification

A specification in this system consists of a specification formula, and the list of *script variable(s)* used in the formula. For example, if we want to detect the bugs

[4] There are a few exceptions such as a node that can accept 2 inputs simultaneously, we address them in 4.2.

```
-> State: 1.1 <-
  ScriptStart1Out = Out
  SetEventMode2In = none
  ...
  MovieClip3State = Stopped
  EventMode = false
-> State: 1.2 <-
  ScriptStart1Out = none
  SetEventMode2In = Enable
-> State: 1.3 <-
  SetEventMode2In = none
  SetEventMode2Out = Out
  EventMode = true
-> State: 1.4 <-
  SetEventMode2Out = none
```

```
  MovieClip3In = Start
-> State: 1.5 <-
  MovieClip3In = none
  MovieClip3State = Playing
-> State: 1.6 <-
  MovieClip3State = Skipped
-> State: 1.7 <-
  MovieClip3Out = Skipped
  MovieClip3State = Stopped
-> State: 1.8 <-
  MovieClip3Out = none
  If5In = In
-> State: 1.9 <-
  If5In = none
  If5Out = False
```

Fig. 6. Value transition of the example control flow

stated in 2.2, the specification can be described as CTL formula fig. 5 (19), and the *script variable* fig. 5 (3). We can expect to verify those kinds of bugs with symbolic model checking by modeling the visual scripts in SMV language.

In our system, users need to write a specification and corresponding node semantics manually. However, users need to write them just once, and after that users can verify scripts automatically. Therefore, we don't think it is a big problem.

3.4 Bug Detection Using Counterexample

As we stated in 3.2, a control flow of a node graph correspond to value transitions of input and output variables. If the property given by `CTLSPEC` is violated, NuSMV generates a counterexample which indicates the witness of property violation. Since the counterexample can be obtained as the form of the value transitions of SMV variables, we can identify the control flow which causes the violation from the counterexample. For example, executing the model in fig.5 by NuSMV outputs a value transition shown in fig. 6. It means that the control flow through Skipped port of Movie Clip node and False port of If node causes violation of the specification. Thus we can detect a bug stated in 2.2.

3.5 Scope and Limitations

Soundness Strictly speaking, the behavior of our model is not exactly the same as the actual behavior of target visual scripts especially from the viewpoint of signal propagation delay. This is because our model needs one state transition to propagate a signal, even though a visual script implementation usually has no delay. For example, in the case of the following 2 signal propagations in fig. 3, the former is faster than the latter in our model, though both of them are the same in visual script implementation. This difference might cause false positive and false negative results of the verification.

– Movie Clip:Finished → Set Event Mode:Disable
– Movie Clip:Skipped → If:True → Set Event Mode:Disable

External Components We also only focus on only behaviors of scripts which are independent of external components. This is because such behaviors of external components are not documented completely and thus it is difficult to model those behaviors. Even if it is difficult to model such behaviors completely, we can partially capture them by assuming that those components behave non-deterministically. For example, the behavior of Movie Clip node in fig. 3 depends on the external components such as "movie player" and "game player input", and we abstract those behavior as non-deterministic state transition (fig. 5 (10)). However, since this assumption allows the model to have non-existent behaviors, it may cause false positive and negative.

State Explosion When the SMV model becomes too large, it is impossible to fully avoid state explosion problem. We address this topic in section 6.

Scope We might be able to avoid the above limitations by more strict modeling. However, since strict modeling can enlarge the model size and causes state explosion easily, we decided to accept these risks. In fact, we currently target the detection of obvious mis-descriptions of visual scripts as stated in section 2.2 and the risk is not a practical problem as far as the result of our preliminary evaluation.

4 Translation Algorithm

4.1 Translation Overview

The procedure that converts a visual script to a corresponding SMV model is shown below with the example of the conversion from the visual script fig. 3 to the SMV model fig. 5. Note that we can implement this conversion as a fully automatic process. However we need to describe specifications and node semantics manually. We explain those issues in section 5.

1. Regarding `VAR` section in SMV models, process the following steps for each node in the visual script:
   (a) Declare an *input* and an *output* variable for the node. Their value domains are `none` and the name of the ports of the node. E.g. Set Event Mode node in fig. 3 has 2 input ports Enable and Disable and 1 output port Out, so the *input* and *output* variables are like fig. 5 (1).
   (b) If the semantics of the node has state transition, declare a *state* variable for the node. E.g. fig. 5 (2) is a *state* variable for the Movie Clip node.
2. Add declaration of *script* variable(s) to `VAR` section according to the specification, e.g. fig. 5 (3).
3. Add `FAIRNESS` constraints for each *state variables*, e.g. fig 5 (4) (see also: section 4.3).
4. Regarding `ASSIGN` section in SMV models, process the following steps for each node in the visual script:

   (a) Convert each input edges of the node to the definition of the *input* variable, e.g. fig 5 (5) (see also: section 4.2).
   (b) Define the *output* variable and the *state* variable by applying the node semantics, e.g. fig. 5 (7)-(11) (see also: 4.4).
5. Add the value transition rules for the *script* variable, e.g. fig. 5 (17)-(18).
6. Insert `SPEC` in SMV models from the specification (fig. 5 (19)).

4.2 Convert Control Flow Edges

In our SMV model, edges in visual scripts are described as definitions of *input variables* as we stated in section 3.2 Control Flow. Consequently, we can convert edges with the following steps:

1. Define the initial value of the *input variable* as `none`, e.g. fig. 5 (12).
2. For each input edge to an input port of the node (from Port1 of Node1 to Port2 of the node), add a rule: `Node1Out = Port1 : Port2`, e.g. fig. 5 (13).
3. Add the default rule that describes the case of no input signal, e.g. fig. 5 (14).

Thus, we can define all the *input variable* according to graph structure of visual scripts automatically.

Handling Simultaneous Inputs As we stated in the section 2.2, more than one node in visual scripts can work in parallel. It means that a node might receive multiple input signals simultaneously. Since only one value can be assigned to an input variable in our model, other input signals are ignored in such case. It might cause an incorrect behavior if some nodes are assumed to handle multiple input signals simultaneously (fortunately these are very rare though).

To avoid this problem, we can declare two input variables for the node whose semantics require to handle two input signals in parallel.

4.3 FAIRNESS constraints

Some node semantics has the nondeterministic assignment for their *state variables* like `MovieClip3State`. This model accepts that it continues to have the value `Playing` infinitely in the context of NuSMV. However, this model is not reasonable, and is expected to finish in a short time. To avoid such a problem, we introduce a fairness constraint which restricts the verification scope to only "fair" state transition. Since our model intends that all nodes eventually return to the initial state, we mechanically add fairness constraints for *state variables* like fig. 5 (4). By adding this constraint, the behavior where the node never returns to the initial state is not considered in verification by NuSMV.

4.4 Apply Node Semantics

Node semantics are given as templates of definition of *output*, *state*(if the node has state transition) and *script variables*, e.g. fig. 7. Note that these definitions only depend on the variables of the node itself, so we can define node semantics independent from graph structure. When our conversion algorithm add definitions of *output* and *state variables* for a node, it selects the appropriate templates for the node and applies them according to the context like the variable names for the node. E.g. there are 2 Set Event Mode nodes, so our conversion algorithm applies the templates (fig. 7) to `SetEventMode2Out` and `SetEventMode4Out` (fig. 5 (6), (15)). However, writing those node semantics as templates is a manual process. We address this issue in section 5.

```
@SetEventMode:define:output_variable
  init(<output_variable>) := none;
  next(<output_variable>) := case
    <input_variable> = Enable | <input_variable> = Disable : Out;
    TRUE : none;
  esac;
@SetEventMode:rule:EventMode
    <input_variable> = Enable | <input_variable> = Enable : true;
```

Fig. 7. An example of node semantics

5 Writing Node Semantics

As we stated in section 4.4, node semantics are described as templates of *output*, *state* and *script variables* definitions. We show how to describe those definitions in this section.

Writing the semantics for every kind of nodes sounds very hard. However, we can classify most of nodes into five types empirically (section 6). Since these semantics are very similar in each class, we can describe node semantics for those classified nodes with a small human cost.

5.1 Output Variables

In our model, value of an output variable describes when and how the node sends output signals. The definition of an output variable is described according to the semantics of the node. E.g. Set Event Mode nodes output signal immediately when they receive input, so the value of `SetEventMode2Out` is changed to `Out` when its input variable `SetEventMode2In` has the value except `none` (fig. 5 (6)). On the other hand, a Movie Clip node output signal after it finishes playing movie, so the value of `MovieClip3Out` is not changed immediately (fig. 5 (7)).

Nondeterministical branch In the case of If node in fig. 3, it branches True or False according to the condition value. A typical approach to model this branch is to decide the output signal non-deterministically since we do not consider data flow and external behavior which affects the condition value. As shown in Fig. 5 (16), the next value of `If5Out` is assigned to True or `False` non-deterministically. Thus, NuSMV verifies the both branch of True and False exhaustively.

5.2 State Transition

Some nodes might have state transition semantics where the node differently behaves for the same stimuli depending on its internal state. To model such a node, we introduce the *state* variable that represents the internal state of the node.

In the case of Movie Clip node in fig. 3, it starts playing the movie clip when it receives an input signal, and then outputs Skipped if the game player skips playing the movie, otherwise it outputs Finished when it finishes playing the movie. With this behavior in mind, we can define the following four states for the state variable `MovieClip3State`:

– `Stopped`: the node is in the initial state.
– `Playing`: the node receives input and playing the movie clip.
– `Finished`: playing movie has finished, and the node sends output through Finished.
– `Skipped`: a game player has skipped playing the movie, and the node sends output through Skipped.

With these states, we can model the semantics of the Movie Clip node with the following steps:

1. The initial state is `Stopped` (fig. 5 (8)).
2. When the node receives the input signal through Start, the state is changed to `Playing` (fig. 5 (9)).
3. When the state is `Playing`, the next state is either `Playing`, `Finished`, or `Skipped` non-deterministically. This description represents the behavior of waiting for completion of the movie playback (fig. 5 (10)).
4. When the state becomes `Finished` or `Skipped`, the node outputs signal through Finished or Skipped respectively (fig. 5 (7)), and the state is back to `Stopped` (fig. 5 (11)).

5.3 Script Variables

Script variables represent variables used in visual scripts and states of external components that visual scripts interact. By defining *Script variables* and describing the conditions for those variables, we can verify those conditions with NuSMV. Note that we need not to define all the variables in visual scripts, but minimum variables that we want to verify in the specification.

Fig. 5 contains a *script variable* `EventMode` that expresses the global flag variable "event mode" in the game system. The value of `EventMode` is defined according to the input of Set Event Mode nodes (fig. 5 (17)). The specification for the script variable can be described in `CTLSPEC` description, we can check the specification stated in 2.2 with NuSMV.

Table 1. Preliminary evaluation of our method

#	# of nodes	# of vars	conv. time[s]	eval. time[s]	detected?
#1	156	356	5.434	192.786	false
#2	94	214	3.878	3.330	false
#3	37	84	1.746	0.056	false
#4	49	119	2.301	0.111	false
#5	177	414	6.625	36.675	true
#6	73	162	2.768	0.173	false
#7	162	408	9.187	98.102	true
#8	430	980	13.286	-	-

Environment: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz / 32GB / Windows 7 Enterprise Service Pack 1 (64 bit) / NuSMV 2.6.0

6 Preliminary Evaluation

For a preliminary evaluation, we implemented a prototype and applied it to the visual scripts that are randomly selected from the scripts used in the production for FFXV. However, we arbitrary selected a very large script only as #8 so that we can identify the limitation on the script size of our method. Table 1 shows the results of the evaluation. The descriptions of these columns are the following:

- #: Script number.
- # of nodes: The number of visual script nodes in the target script.
- # of vars: The number of SMV variables in the generated SMV model.
- conv. time: Conversion time from the visual script to the SMV model with our method. We tried 5 times for each script and adopted a median value of those trial.
- eval. time: Execution time of NuSMV for the model. We tried 5 times for each script and adopted a median value of those trial.
- detected?: Whether NuSMV detected a problem in the script or not.

Node Semantics We prepared an encoding by SMV language for each node in the scripts. As stated in section 5, we can straightforwardly prepare an encoding for nodes with simplified semantics. The eight scripts shown in Table 1 have 164 kinds of nodes, and they are classified as follows:

1. single output: 98 kinds of nodes.
2. multi-outputs with conditions (non-deterministic choice): 7 kinds of nodes.
3. multi-outputs with state-transition: 14 kinds of nodes.
4. multi-outputs with conditions (non-deterministic choice) and state-transition: 12 kinds of nodes.
5. entry point: 3 kinds of nodes.
6. node with custom semantics: 30 kinds of nodes.

30 kinds of nodes have custom semantics and we manually prepared encodings for them. However, we can mechanically translate the 134 kinds of nodes (82%) which are classified to 1) to 5) into the SMV model. This result demonstrates that our translation method has enough availability in practical use.

Results Regarding precision, our method found counterexamples on two scripts during the preliminary evaluation. We confirmed with the game designers that the counterexamples are not false positives. [5] This result demonstrates that our method can detect the specific types of bugs that we are focusing on.

Regarding recall, we also checked these scripts by visual inspection. As long as our inspection, there was no false negative.

Limitation It appeared that our algorithm cannot handle very large scripts, since the verification of #8 had not finished within 3 hours. Improving our algorithm to handle those large scripts is future work.

7 Summary and Future Work

We described an automatic verification method for node-based visual script notation for efficient game production. Our method automatically converts visual script implementation to the input model for NuSMV. We confirmed through a preliminary evaluation that our method can detect the specific types of bugs that we are focusing on in realistic time on most of the visual scripts used in the production for FFXV.

A next step for extending this work would be compositional verification [2]. It appears that there are some very large scripts used in the production for FFXV, that our method cannot handle. If we can split the model and verify those sub-models separately, we can reduce the exponential order of the verification and expect that those verifications can be handled in a reasonable time. Also, if we can verify more than one script together, we can track the control flow across the scripts and can expect to reduce false positives/negatives. Compositional verification might make it possible to verify multiple models too. Another next step would be the automated generation of node semantics. Currently, we need to write node semantics manually. If we can extract semantics from node implementation, we can increase the range of automation of our method.

Acknowledgment

We wish to thank the collaborative researchers for helpful discussions. We also wish to thank the FINAL FANTASY XV development team for supporting our research. UNREAL ENGINE is a trademark or registered trademark of Epic Games, Inc. Windows is a trademark or registered trademark of Microsoft Corporation. All other trademarks are the property of their respective owners.

[5] According to the game designers, those scripts are used only in the trial version, so they will not fix the bugs though.

References

1. Ben-Ari, M., Pnueli, A., Manna, Z.: The Temporal Logic of Branching Time. Acta Informatica **20**(3), 207–226 (1983)
2. Berezin, S., Campos, S., Clarke, E.M.: Compositional reasoning in model checking. In: Proc. of Int'l Symp. on Compositionality (COMPOS'97). pp. 81–102 (1998)
3. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hawng, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. Information and Computation **98**(2), 142–170 (1992)
4. Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.D.: Model checking large software specifications. IEEE Trans. Softw. Eng. **24**(7), 498–520 (1998)
5. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Verifier. In: The 11th International Conference on Computer Aided Verification (CAV' 99) , LNCS1633. pp. 495–499 (1999)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT press (1999)
7. FINAL FANTASY XV, http://www.jp.square-enix.com/ff15/
8. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming **8**(3), 231 – 274 (1987)
9. Hasegawa, I., Nozoe, R., Ono, T., Koyama, M., Ishida, T.: Visual effects of final fantasy xv: Concept, environment and implementation. In: ACM SIGGRAPH 2016 Talks. pp. 23:1–23:2. SIGGRAPH '16, ACM, New York, NY, USA (2016)
10. Jussila, T., Dubrovin, J., Junttila, T., Latvala, T., Porres, I., Kepler, J., Linz, U.: Model Checking Dynamic and Hierarchical UML State Machines. In: Proc. of the 3rd Int'l Workshop on Model Development, Validation and Verification (MoDeVa ' 06). pp. 94–110 (2006)
11. Leveson, N.G., Heimdahl, M.P.E., Hildreth, H., Reese, J.D.: Requirements Specification for Process-Control Systems. IEEE Transactions on Software Engineering **20**(9), 684–707 (1994). https://doi.org/10.1109/32.317428
12. Lumberyard Script Canvas, https://docs.aws.amazon.com/lumberyard/latest/userguide/script-canvas-introduction.html
13. Moreno-Ger, P., Fuentes-Fernández, R., Sierra-Rodríguez, J.L., Fernández-Manjón, B.: Model-checking for adventure videogames. Information and Software Technology **51**(3), 564–580 (2009)
14. Pnueli, A.: A temporal logic of concurrent programs. Theoretical Computer Science **13**, 45–60 (1981)
15. Radomski, S., Neubacher, T.: Formal Verification of Selected Game-Logic Specifications. In: Proc. of the 2nd EICS Workshop on Engineering Interactive Computer Systems with SCXML. pp. 30–34 (2015)
16. Rezin, R., Afanasyev, I., Mazzara, M., Rivera, V.: Model checking in multiplayer games development. In: 2018 IEEE 32nd Int'l Conf. on Advanced Information Networking and Applications (AINA). pp. 826–833 (2018)
17. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education (2004)
18. Unreal Engine 4 Blueprints, http://api.unrealengine.com/INT/Engine/Blueprints/index.html
19. Zhao, Q., Krogh, B.H.: Formal verification of statecharts using finite-state model checkers. IEEE Trans. on Control Systems Technology **14**(5), 943–950 (2006)