IADIS International Journal on Computer Science and Information Systems
Vol. 13, No. 1, pp. 82-95
ISSN: 1646-3692

ACTIVITY ANALYSIS (ACTA): EMPOWERING SMART GAME DESIGN WITH A GENERAL PURPOSE FSM DESCRIPTION LANGUAGE

Emmanouil Zidianakis, Margherita Antona and Constantine Stephanidis
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), N. Plastira 100, Heraklion, GR-70013, Greece

ABSTRACT

Game design process creates goals, rules, and challenges to define a game that produces desirable interactions among its participants. Modeling the way in which players interact with the game requires among others the definition of game rules, plot, connection between player and the game, etc. ACTA (ACTivity Analysis) is a general purpose finite state machine (FSM) description language for rapid prototyping of smart games. ACTA facilitates smart game design by intervention professionals who are not familiar with traditional programming languages. Professionals are able to model the game's logic and interaction using a straightforward and intuitive notation which can be customized for every different game. In order to support a customized scripting vocabulary, a reflection-oriented model was adopted allowing dynamic linking between ACTA generated rules and multiple data types. Windows Workflow Foundation (WF) is the "glue layer" connecting ACTA's scripting vocabulary with the application internals. Moreover, developers can use ACTA not only for developing event-driven sequential logic games, but also for applications of behavior composed of a finite number of states, transitions between those states and actions, as well as for applications based on rules driven workflows. As a case study illustrating the usefulness and benefits of ACTA, a body posture imitation game, called Mimesis game, was developed based on occupational therapy expertise and practice.

KEYWORDS

Smart Games Rapid Prototyping, Finite State Machine Interaction Modeling

1. INTRODUCTION

There are many ways of modeling the behavior of systems, and the use of state machines is one of the oldest and best known. Finite state machines (FSMs) are commonly used to organize and represent an execution flow. In games, they are most known for being used in AI, but they are also common in implementations of user input handling, navigating menu screens, parsing text, and other asynchronous behavior [Buckland 2005]. According to the literature, there are a number of techniques for implementing a FSM. One of the most common methods is nested switch case statements. Although this technique is common, it is not easy to maintain as the FSM grows. Alternatively, a variety of FSM modeling tools is available, capable of representing a large logic by a relatively small diagram. The majority of them seem to follow simple, easy to verify rules and can be used to generate code.

Smart game design is a domain where FSMs are widely employed, often by non-experts such as teachers, intervention professionals and therapists. Working with FSM modeling tools based on specific syntax and complex notations creates challenges for non-developers when they need to model even a simple control execution flow. For the purposes of the present work, a minimal finite state machine description script language (ACTA) was developed in the context of interaction modeling for smart game design, based on a simplified notation and a straightforward and intuitive syntax. According to [Zidianakis et al 2015], traditional games are turned into smart games that share unique characteristics such being adaptive to players' skills and abilities. The aim of ACTA is to facilitate the smart game design process by professionals who are not familiar with traditional programming languages. Professionals are able to model the game's logic and interaction using a simplified notation which can be customized for every different game. Furthermore, developers can use ACTA not only for developing event-driven sequential logic games, but also for applications of behavior composed of a finite number of states, transitions between those states, and actions as well as for applications based on rules driven workflows.

2. BACKGROUND AND RELATED WORK

According to the literature, many related efforts have primarily focused on the description of FSMs. The Unified Modeling Language (UML) is an indicative example as it has an exuberant notation for describing FSMs. Some concrete examples are MetaUML [Gheorghies 2009], yUML [Haris 2009], and TextUML [Chaves 2009], which primarily focus on the creation of visual models based on textual notation. The majority of textual modeling tools support automated code generation. For example, Ragel supports a number of high-level languages but is targeted towards text parsing and input validation [Thurston 2009]. Ruby on Rails has built-in support for FSMs. The State Machine Compiler (SMC) [Charles 2009] is targeted towards the specification of event driven systems. Microsoft has also developed a textual specification language called Abstract State Machine Language (AsmL) [Gurevich et al 2005], based on concepts of the theory of abstract state machine. Executable UML [Milicev 2009] supports a subset of UML textually but misses key features of UML and does not integrate with programming languages. Umple [Badreddin 2010] merges the concepts of programming and modeling by adding modeling abstractions directly into programming languages.

According to [Kleppe et al 2003], modeling is considered to be a good widely accepted software development practice. However, as stated in [Forward et al 2010], modeling practices are not as widely adopted as they could be, primarily due to: a) the low adoption of modeling tools caused by their heavyweight nature, b) poor generated code, and c) difficulty between code and model synchronization. In addition, most modeling tools are intended to be used only by developers either using purely textual approaches or editing diagrams. With respect to the aforementioned modeling practices, the majority of modeling tools aim at providing a generic solution of modeling any-type of FSM in a context free manner. In addition, most modeling tools are intended to be used only by developers familiar with either using purely textual approaches or editing diagrams. As a result, working with FSM modeling tools based on specific syntax and complex notation, creates challenges faced by non-developers when they need to model even a simple control execution flow. For the purposes of the present work, a minimal finite state machine description script language, called ACTA, was developed based on a straightforward and intuitive syntax. The aim of ACTA (ACTivity Analysis) is to facilitate activity analysis process during smart game design by professionals who are not familiar with traditional programming languages, by providing a simplified notation. ACTA comes with an IDE which fully supports its use.

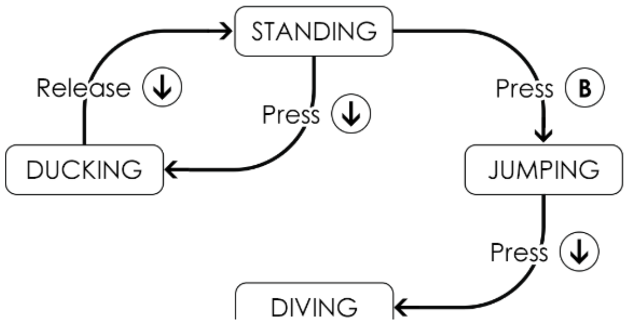
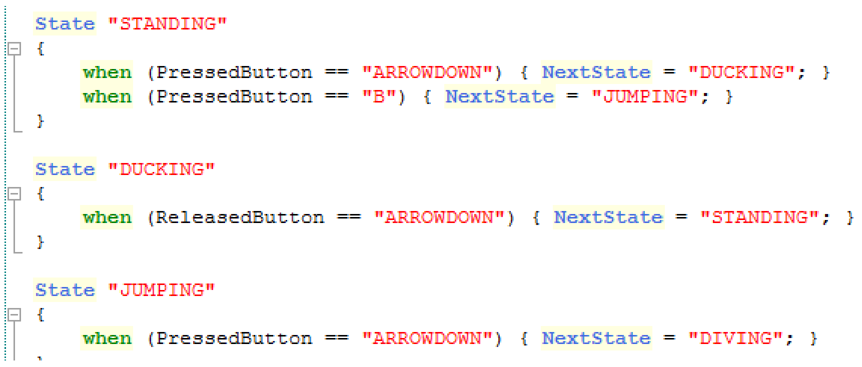


Figure 1. A Finite State Machine example

3. ACTA: A GENERAL PURPOSE FINITE STATE MACHINE (FSM) DESCRIPTION LANGUAGE FOR SMART GAME DESIGN

ACTA aims at facilitating smart game design by providing the authors with a straightforward and intuitive syntax to describe gameplay attributes such as states, inputs and rules triggering transition among the states. A script in ACTA can describe a FSM as depicted in the example of Figure 1, which illustrates a flowchart of a hero game. The flowchart consists of boxes for each action a hero can do such as standing, jumping, ducking, and diving. When acting, the hero can respond to a button press and perform a different action. Furthermore, in each of the depicted boxes, there are arrows connecting the current action with a next one according to the labeled pressed button. An example of the hero game in ACTA script is depicted in Figure 2. In detail, the ACTA script of this example consists of the following parts: a) a fixed set of states that the machine can be in (e.g., standing, jumping, etc.), b) the current state, c) a sequence of inputs or events (e.g., the raw button presses and releases), and d) a set of transitions from the current state to a new one based on incoming input or event. When an input or event occurs, if it matches a transition for the current state, the machine changes to the state that transition points to.



```
State "STANDING"
{
    when (PressedButton == "ARROWDOWN") { NextState = "DUCKING"; }
    when (PressedButton == "B") { NextState = "JUMPING"; }
}

State "DUCKING"
{
    when (ReleasedButton == "ARROWDOWN") { NextState = "STANDING"; }
}

State "JUMPING"
{
    when (PressedButton == "ARROWDOWN") { NextState = "DIVING"; }
```

Figure 2. Finite states of an action game in ACTA script

3.1 ACTA's Syntax and Semantics

Authors can use ACTA for scripting a smart game or application in general, by outlining a set of finite states and event-driven transitions (i.e., a transition from one state to another is triggered by an event or a message). Thus, ACTA facilitates event driven programming, in which the flow of the program is determined by events, event handlers, and asynchronous programming.

As depicted in Figure 3, a script in ACTA initially consists of various transitions that may happen anytime during execution without caring about the current state (i.e., *when* statements). For example, when a user gets into the playing room, the game starts by switching the machine to the "init" state to initialize the game. Another example is that during play, when the user leaves the room, the machine goes to the "paused" state to handle this situation. In this state, the game gets paused until the user comes back (within a period of *N* seconds), returning the machine to the previous state. When the allowed time has elapsed, the game terminates by switching to the corresponding "end" state. The ACTA script continues with the definition of the states (i.e., State statements).

ACTA uses three (3) fundamental keywords for scripting a finite state machine: a) *State*, b) *NextState*, and c) *when*. Table 1 summarizes all keywords used in ACTA scripts. The *NextState* keyword is used to set a new state. Using an assignment expression, the author simply sets the *id* of the next state to the reserved word *NextState* (e.g., *NextState* = *"state id"*). Event-driven transitions are described using the keyword *when*. In case a *when* statement's condition becomes true, a transition from the current state to a new one is fired. As depicted in Figure 3, within the when statement's block there are optional macros to be executed when exiting the state. To point to the new state, authors define a *NextState* statement at the end of the block. A finite state is described using the keyword *State*. In detail, a *State* statement defines a block that contains: a) optional macros for execution when entering the state, and b) transitions. The latter may be either an explicit state transition using the keyword *NextState* or a set of when statements in order for the machine to remain in the current state until the condition of a *when* statement becomes true and fires its transition.

Using ACTA, authors are able to set the execution order of a *when* statement by setting its priority accordingly. *When* statements that have a higher priority value are executed first. The default value for any *when* statement is *0*. Furthermore, authors are able to characterize a *when* statement with a unique *id* by setting its *Name* property. So, *when* statements that have the same priority are executed in the alphabetic order of their name properties.

Table 1. ACTA keywords

`State`	`when`	`with`	`NextState`
`Name`	`Priority`	`Reevaluation`	`Active`
`true`	`True`	`false`	`False`
`null`	`Null`	`priority`	`name`
`active`	`Never`	`reevaluation`	`PreviousState`
`When_SetActive`	`When_SetReevaluation`	`When_SetPriority`	

Additionally, authors can set the re-evaluation behaviour of a *when* statement. In detail, they can specify whether a *when* statement can be re-evaluated by setting *Always* or *Never* to the *Revaluation* property. *Revaluation* behaviour is used to limit the number of times that a *when* statement runs, primarily to prevent infinite loops (i.e., state A → B → C → A etc.). *Always* (which is the default value) indicates that the *when* statement can be re-evaluated multiple times. *Never* indicates that the *when* statement is executed only once. A condition may be evaluated several times until the *when* statement executes macros, but the *when* statement will never be evaluated again. Also, authors are able to disable a *when* statement by setting the value of its *Active* property to *True* (default) or *False***.** *Active* property indicates whether the *when* statement should be evaluated. Setting to *False* this property is similar to comment out the *when* statement.

Finally, authors have access to a set of library functions to dynamically change the semantics of *when* statements. This set consists of the following:

- *When_SetActive(bool active, params string[] whenNames):* sets the *Active* property of *when* statements with names presented into the list *whenNames* to the value given as a parameter (i.e., *active*).
- *When_SetReevaluation (string reevaluation, params string[] whenNames):* sets the *Revaluation* behaviour property of *when* statements with names presented into the list *whenNames* to the first parameter's value.
- *When_SetPriority (int priority, params string[] whenNames):* sets the *Priority* property (i.e. execution order) of *when* statements with names presented into the list *whenNames* to the first parameter's value.

The example, depicted Figure 4, consists of two *when* statements of the same priority. Initially, the second *when* statement (i.e., *when#2*) is not active. When the value of the subtotal field becomes greater than *10000*, *when#1* executes its macros. The first one (i.e., *When_SetActive*) activates *when#2*, while the second one, sets the discount field equal to 0.5. So, *when#1* reads the subtotal field and writes to the discount field, while *when#2* reads the discount and subtotal fields and writes to the total field. Therefore, *when#2* is active and has a dependency on when#1. As a result, *when#2* is evaluated/reevaluated whenever *when#1* executes its macros.

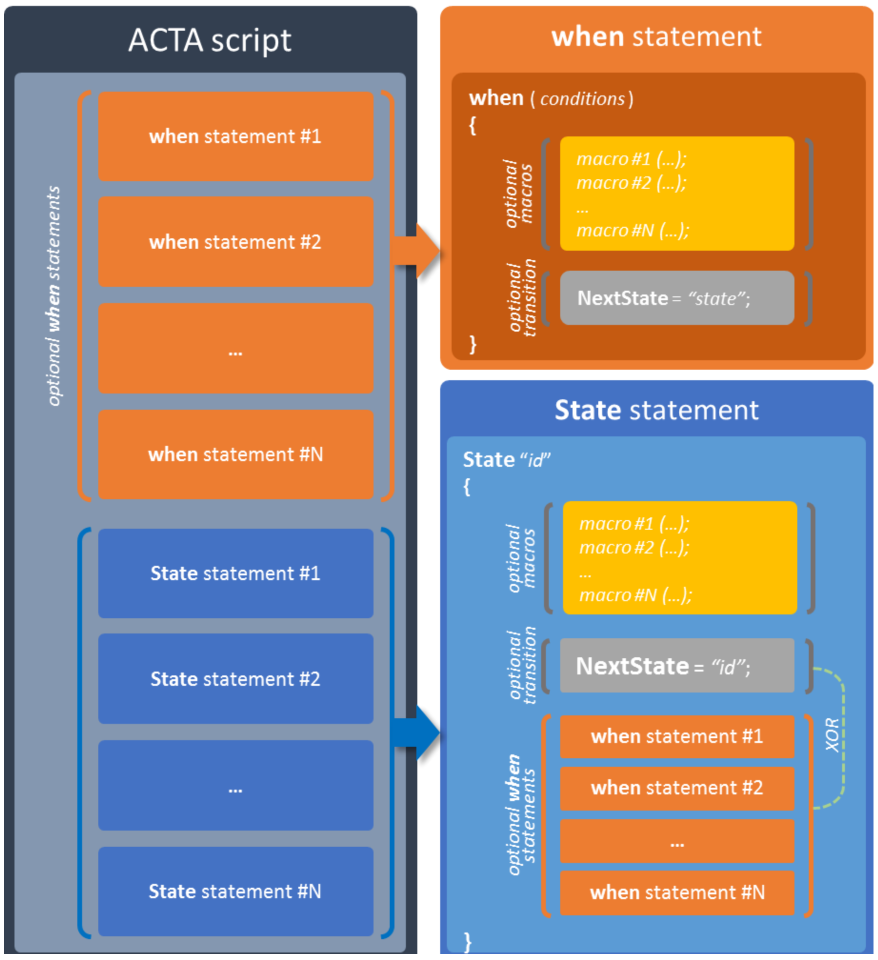


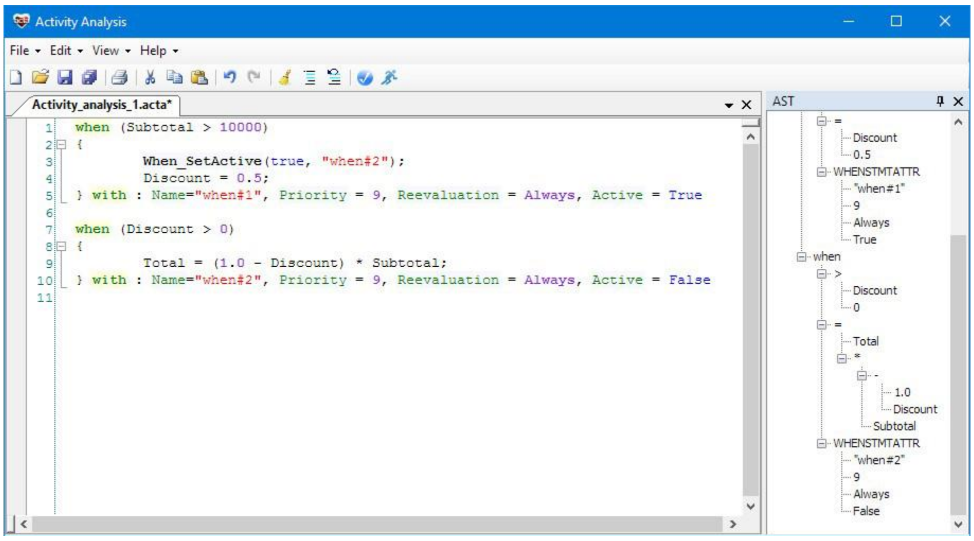
Figure 3. ACTA's structure

3.2 ACTA from a Developer's Perspective

Windows Workflow Foundation[1] (WF) is the "glue layer" connecting ACTA's scripting vocabulary with the application internals. WF is a Microsoft technology that provides an in-process workflow engine to implement workflows within .NET applications. Furthermore, WF provides rule capabilities to the .NET Framework development platform. These capabilities range from simple conditions that drive activity execution behavior to complex rulesets executed by a full-featured forward-chaining rules engine. To this end, an ACTA script is converted to a WF ruleset in order to take advantage of the provided workflow rule engine supporting complex rules scenarios, demanding forward chaining evaluation and precise evaluation control. Figure 4 illustrates the ACTA IDE which provides comprehensive

[1] https://msdn.microsoft.com/en-us/library/vstudio/ms734631(v=vs.90).aspx

facilities to programmers for scripting in ACTA and generate the corresponding ruleset (filename extension *.rules*). A WF ruleset is a collection of rules with a set of execution semantics (e.g., *If-Then-Else* expressions that operate on properties within application). The generated rule expressions are given to the WF rule engine for evaluation (Figure 5).



```
when (Subtotal > 10000)
{
        When_SetActive(true, "when#2");
        Discount = 0.5;
} with : Name="when#1", Priority = 9, Reevaluation = Always, Active = True

when (Discount > 0)
{
        Total = (1.0 - Discount) * Subtotal;
} with : Name="when#2", Priority = 9, Reevaluation = Always, Active = False
```

```
      =
        Discount
        0.5
      WHENSTMTATTR
        "when#1"
        9
        Always
        True
    when
      >
        Discount
        0
      =
        Total
        *
          -
            1.0
            Discount
          Subtotal
      WHENSTMTATTR
        "when#2"
        9
        Always
        False
```

Figure 4. ACTA IDE

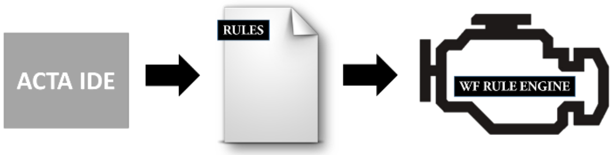


Figure 5. WF rule engine takes the ACTA generated ruleset for evaluation

The ACTA IDE parses the statements found in an ACTA script and generates the equivalent ruleset through three distinct steps. Firstly, it parses ACTA script and translates each statement or expression found to its equal code expression using the Code Document Object Model (CodeDOM[2]). The .NET Framework includes this mechanism that enables developers to emit source code to generate source code in multiple programming languages at run time, based on a single model that represents the code to render. In detail, a WF rule is created for each CodeDOM expression by defining a condition (i.e., *rule expression condition*) with an associated set of actions to perform (i.e., *then actions*). For example, a logical expression in ACTA (e.g., x==5) is converted to an instance of CodeBinaryOperatorExpression with operator type of CodeBinaryOperatorType. ValueEquality (see Figure 6). Secondly, a WF rule is created for each CodeDOM expression

2 https://msdn.microsoft.com/en-us/library/650ax5cx(v=vs.110).aspx

by defining a condition (i.e., rule expression condition) with an associated set of actions to perform (i.e., then actions). Finally, the ACTA IDE creates a ruleset of the produced rules and uses WorkflowMarkupSerializer to write the ruleset to file. WorkflowMarkupSerializer provides methods that serialize workflow classes to XAML format.

```
<CodeBinaryOperatorExpression Operator="ValueEquality">
 <CodeBinaryOperatorExpression.Right>
  <CodePrimitiveExpression>
    <CodePrimitiveExpression.Value>
      <Single>5</Single>
    </CodePrimitiveExpression.Value>
 </CodePrimitiveExpression>
</CodeBinaryOperatorExpression.Right>
<CodeBinaryOperatorExpression.Left>
  <CodeCastExpression TargetType="System.Single">
    <CodeCastExpression.Expression>
      <CodeMethodInvokeExpression>
        <CodeMethodInvokeExpression.Parameters>
          <CodePrimitiveExpression>
            <CodePrimitiveExpression.Value>
              <String >x</String>
            </CodePrimitiveExpression.Value>
          </CodePrimitiveExpression>
        </CodeMethodInvokeExpression.Parameters>
        <CodeMethodInvokeExpression.Method>
          <CodeMethodReferenceExpression MethodName="GetProperty">
                 <CodeMethodReferenceExpression.TargetObject>
                   <CodeThisReferenceExpression />
              </CodeMethodReferenceExpression.TargetObject>
            </CodeMethodReferenceExpression>
          </CodeMethodInvokeExpression.Method>
        </CodeMethodInvokeExpression>
      </CodeCastExpression.Expression>
    </CodeCastExpression>
 </CodeBinaryOperatorExpression.Left>
</CodeBinaryOperatorExpression>
```

Figure 6. Rule condition expression using CodeDOM

In summary, ACTA is a general purpose finite state machine description language that facilitates the development of .NET applications by integrating into their projects the Windows Workflow Rule engine. In Windows Workflow Foundation all rulesets are defined in relation to a single specific .NET type. In case of ACTA, scripting such a .NET type is not known in advance. To overcome this difficulty and allow dynamic linking between ACTA generated rules and multiple types, a reflection-oriented model was adopted and implemented as a .NET framework library. To this end, variables used in ACTA are mapped to properties considered public to reflection (a property must have at least one accessor that is public). For example, the logical expression ***x==5***, part of the when statement (e.g., *when(**x==5**) { NextState = "A" }*), is equal to the following condition in CodeDOM: *(float)this.GetProperty("x") == 5f* .

3.3 Implementation Details

The ACTA IDE was implemented in .NET using C# and uses a parser generated with ANTLR[3] to read and process ACTA scripts. ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing and translating structured text or binary files. It is widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees. Taking as input the ACTA grammar (see the example in Figure 7), ANTLR generates a lexer and a parser for that language that can automatically build parse-trees, which are data structures representing how a grammar matches the input. ANTLR also automatically generates tree walkers that can be used to visit the nodes of those trees to execute application-specific code. The ACTA IDE provides authors with SyntaxBox[4], a syntax highlight editor. SyntaxBox features powerful syntax highlight windows forms control for the Microsoft.NET Platform. It supports: a) syntax highlighting, b) code folding for just about any programming language, c) bracket matching, d) unlimited undo/redo buffer, e) bookmarks, etc.

```
program : whenstmt* mystate* EOF -> ^( GLB_STATELIST whenstmt* mystate*);

mystate : 'State' STRING '{' stmt2* whenstmt* '}'-> ^(MYSTATE STRING stmt2*
whenstmt*);

whenstmt : WHEN '(' logicalExpression ')' '{' stmt2* '}' whenstmtattr? ->
^(WHEN logicalExpression stmt2* whenstmtattr?);

whenstmtattr: 'with' ':' whenid ',' priority ',' reevaluation (',' whenactive)?
-> ^(WHENSTMTATTR whenid priority reevaluation whenactive?);

whenid : ('Name' | 'name')! ASS! STRING;

priority : ('Priority' | 'priority')! ASS! NUM;

reevaluation: ('Reevaluation' | 'reevaluation')! ASS! reevaluationAttr;

reevaluationAttr: 'Always' | 'Never';

whenactive: ('Active' | 'active')! ASS! BOOLEAN;
```

Figure 7. An example of ACTA grammar in ANTLR

4. MIMESIS GAME: A CASE STUDY

As a case study illustrating the usefulness and benefits of ACTA, this section presents the Mimesis game, which has been developed in ACTA. The Mimesis game is an interactive game for preschool children based on occupational therapy (OT) expertise and practice. The game aims to unobtrusively monitor, evaluate and enhance child's postural praxis. Praxis, also called as motor planning, refers on how human brain plans for and carries out movements that have not done before. The game can automatically adapt at run time in order to meet the

[3] http://www.antlr.org/

[4] https://code.google.com/p/alsing/wiki/SyntaxBox

estimated needs of children by providing proper personalized support. This is achieved by applying scaffolding strategies according to which, appropriate amounts of support should be given at proper times during playing in order to support young children skills and abilities [Cole et al 1978].

Using a sensory infrastructure presented in [Zidianakis et al 2014], the game measures the quality and performance of the body posture that the child assumes and can extract indications of the achieved maturity level and skills of the child. Regarding the gameplay, the child stands in front of the interactive display, which is positioned horizontally, (e.g., near to a wall) giving the feeling of standing in front of a mirror. At the upper side of the large display, a depth camera (i.e., Kinect sensor) is positioned, which allows the sensory infrastructure to recognize the user's position and gestures. The Mimesis game requires the child to imitate a series of postures presented by a virtual avatar called Max [Zidianakis et al 2014], a process that requires motor planning or programming a skilled, non-habitual motor action. Moreover, when Max utters verbal commands, the child's abilities related to understand and follow verbal directions and to plan movements based on verbal instructions are also stimulated.

The game includes several body postures that are presented by the virtual avatar one at a time (see Figure 8). Indisputably, it is required close observation of the posture, motor planning, and movement execution. Children are instructed to perform the imitation as quickly as possible. According to the Postural Praxis test [Ayres 1980], in order to perform well, children need good sensation of their own body position. This imitation skill is necessary for participating in games such as "Follow the Leader" and in learning specific sports moves. Many early childhood classrooms include imitation games at circle time to music or rhymes.

The Mimesis gameplay was scripted in ACTA by early intervention professionals as depicted in Figure 10. The game consists of the following functions: a) Max presenting various poses, b) Max assuming various postures (moving from his idle position to a posture), c) voice commands of each pose, and d) audio description of each pose.

4.1 Starting Point of the Game

As illustrated in Figure 10, when the child enters the camera view area, Max welcomes the child by saying "*Hello. I'm Max. Do you want to play?*". In this case, a state transition occurs from the state *Idle* to the state *Welcome*. While remaining in the last state, if the child answers *YES*, the control flow is transferred to the state *Mimesis* where Max asks the child to imitate a pose by saying "*Do what I do as fast as you can*". Similarly, if the child answers *NO*, then Max looks sad and says "*Goodbye*" (state *End*). If the child does not react to that question within a period of time (e.g. elapsed time of 15 sec.), Max welcomes him again (next state *Welcome*). Anytime the child moves away from the camera's view area (global when condition), Max says "*You are leaving me*" and then looks sad and says "*Goodbye*". If the child returns after a period of time, Max welcomes again.

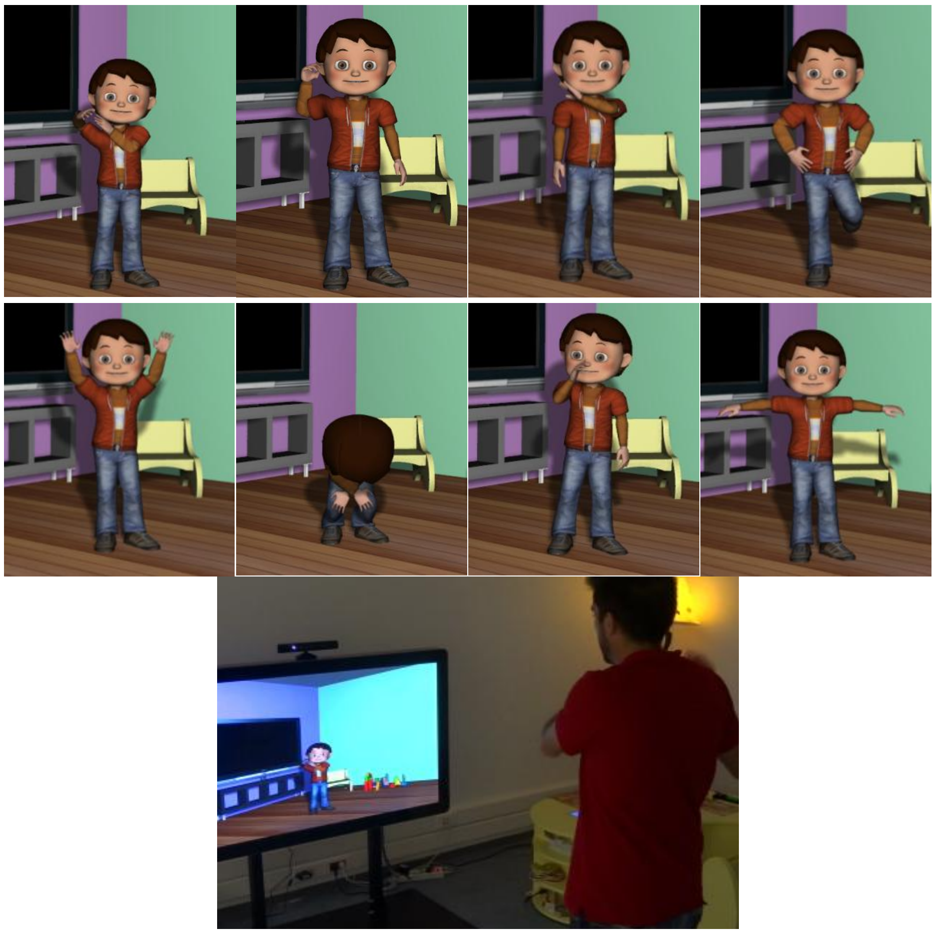


Figure 8. Postures demonstrated by Max in the Mimesis game

4.2 Game Levels

The Mimesis game consists of four game levels. At the first game level, Max demonstrates a randomly selected static pose without giving instructions (verbal information). For example, Max puts his hand on his nose instantly without tracking his movement and without describing what he does. If the child reacts properly within a certain period of time, Max congratulates the child by saying a random message such as, “Congratulations”, “Bravo”, “Very well”, etc. Thereafter, the game continues as presented in “Playing with Max” section (see 4.3). In the case that the child does not respond properly within a given period of time, the game proceeds to the second level.

At the second game level, Max repeats the gesture in which the child did not respond properly at the first level, without giving instructions (verbal information). For example, Max puts his hand on his nose while the child can track his movement. Similarly, Max does not describe what he does. If the child reacts properly within a given period of time, Max congratulates the child and continues as presented in “Playing with Max” section (see 4.3). If the child does not respond properly within a given period of time, the game proceeds to the third level.

At the third game level, Max demonstrates the gesture in which the child did not respond properly at the second level, with instructions (verbal information). For example, Max puts his hand on his nose instantly while describing what he does. If the child reacts properly within a given period of time, Max congratulates the child and the game continues as presented in “Playing with Max” section (see 4.3). In the case that the child does not respond properly within a certain period of time, the game proceeds to the forth level.

At the fourth game level, Max imitates the gesture in which the child did not respond properly at the third level, with a verbal command prompting the child to follow. For example, Max puts his hand on his nose while the child can track his movement and he says “put your hand on your nose”. If the child reacts properly within a given period of time, Max congratulates the child. If the child does not react properly within a the set period of time, the game continues as presented in “Playing with Max” section (see 4.3) with the exception that if it is the 3rd consecutive time that the games passes to the fourth level the smart game ends with Max saying “goodbye”.

4.3 Playing with Max

The Mimesis game randomly selects a different posture and the procedure continues from the first level. When all postures have appeared the game ends with Max saying “thanks” to the player and starting the presentation of a short compilation video of photographs taken during playing (see Figure 9 for some screenshots).

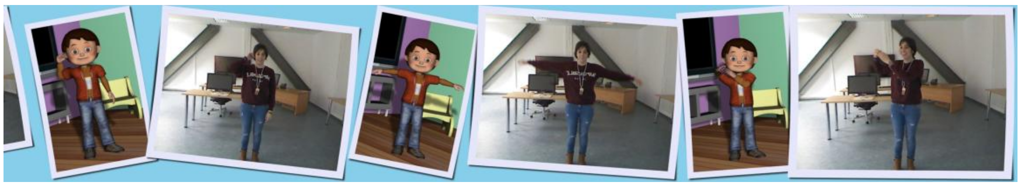


Figure 9. A short compilation video of photographs taken during Mimesis game

```
when(ChildPresent && !SessionIsActive) { NextState = "Idle"; }

when(!ChildPresent && SessionIsActive) {
    MaxSays ("You are leaving me");
    NextState = "ChildIsLeaving";
}

when(UserTooFar && SessionIsActive) {
    SpeakAsync("Please, come closer to me!.");
    NextState = "ChildIsTooFar";
}

State "Idle" {
    when(ChildPresent) {
        StartSession();
        NextState = "Welcome";
    }
}

State "Welcome" {
    MaxSays("Hello. I'm Max. Do you want to play? ");
    when(SpeechRecognized == "yes") { NextState = "Mimesis"; }
    when(SpeechRecognized == "no") { NextState = "End"; }
    when(Time == 15 && Count == 1) { NextState = "Welcome"; }
    when(Time > 15 && Count != 1) { NextState = "End"; }
}
State "End" {
    MaxSays ("Goodbye!");
    EndSession();
    when(Time == 10) { NextState = "Idle"; }
}
State "Mimesis" {
    MaxSays ("Do what I do as fast as you can");
    ShufflePoses();
    when(Time == 5) { NextState = "Level1"; }
}
State "Level1" { //nonverbal communication
    DemonstrateStaticPose(CurrentPose);
    when(AnimationRecognized == CurrentPose) { NextState = "Congrats"; }
    when(Time == 10) { NextState = "Level2"; }
}
State "Congrats" {
    MaxSays (GetRandomExclamation);
    NextPose();
    when(!PosesDidFinish) { NextState = "Level1"; }
    when(PosesDidFinish) { NextState = "Reward"; }
}
```

Figure 10. ACTA script snippet used in Mimesis game

5. CONCLUSION

A general purpose finite state machine (FSM) description language for rapid prototyping of smart games based on the outcome of the activity analysis (ACTA) process held by intervention professionals. ACTA facilitates smart game design by professionals who may need to design smart games but are not familiar with traditional programming languages. Smart game developers can use ACTA not only for developing event-driven sequential logic games, but also for applications of behavior composed of a finite number of states, transitions between those states, and actions as well as for applications based on rules driven workflows. ACTA facilitates the development of .NET applications by integrating into the Windows Workflow Rule engine. The ACTA IDE provides comprehensive facilities to authors and programmers for scripting in ACTA and generates the corresponding ruleset which in turn is given to the WF rule engine for evaluation. The Mimesis Game, a case study, was presented for demonstration purposes. The Mimesis gameplay was scripted in ACTA by early intervention professionals. The Mimesis game aims at promoting the child's posturing ability by challenging him/her to imitate a series of demonstrated postures. Regarding future work, ACTA capabilities will expand so as authors to be able to set alternative sequential finite states based on some condition. Finally, it is considered crucial to design an evaluation strategy with the active contribution of the aforementioned end users (e.g., teachers, therapists, child development experts) in order to measure ACTA's applicability and usability from an end-users perspective.

ACKNOWLEDGEMENT

This work is supported by the FORTH-ICS internal RTD Programme 'Ambient Intelligence and Smart Environments'.

REFERENCES

Ayres, A.J., 1980. *Southern California sensory integration tests manual*. Western Psychological Services, pp.35-43.

Badreddin, O., 2010, May. *Umple: a model-oriented programming language*. In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2 (pp. 337-338). ACM.

Buckland, M., 2005. Programming game AI by example. Jones & Bartlett Learning.

Charles, R. 2009. "The state machine compiler". http://smc.sourceforge.net/

Chaves, R. 2009. "TextUML". http://abstratt.com/

Cole, M. and Scribner, S., 1978. *The development of higher psychological processes*.

Forward, A., Badreddin, O. and Lethbridge, T.C., 2010, June. *Perceptions of software modeling: a survey of software practitioners*. In 5th workshop from code centric to model centric: evaluating the effectiveness of MDD (C2M: EEMDD).

Gheorghies, O. 2009. "MetaUML". http://metauml.sourceforge.net/

Gurevich, Y., Rossman, B. and Schulte, W. 2005. *"Semantic essence of AsmL"*. Th Computer Science, 343, 370-412.

Haris, T. "yUML". 2009. http://www.yuml.me/

Kleppe, A.G., Warmer, J.B. and Bast, W., 2003. *MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Professional.

Milicev, D., 2009. *Model-driven development with executable UML*. John Wiley & Sons.

Thurston, A. 2009. "Ragel state machine compiler". http://www.complang.org/ragel

Zidianakis, E., Ioannidi, D., Antona, M. and Stephanidis, C., 2015, November. *Modeling and Assessing Young Children Abilities and Development in Ambient Intelligence*. In European Conference on Ambient Intelligence (pp. 17-33). Springer International Publishing.

Zidianakis, E., Papagiannakis, G. and Stephanidis, C., 2014, November. A cross-platform, remotely-controlled mobile avatar simulation framework for AmI environments. In SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications (p. 12). ACM.

Zidianakis, E., Partarakis, N., Antona, M. and Stephanidis, C., 2014, June. *Building a sensory infrastructure to support interaction and monitoring in ambient intelligence environments.* In International Conference on Distributed, Ambient, and Pervasive Interactions (pp. 519-529). Springer International Publishing.