Journal of Universal Computer Science, vol. 13, no. 6 (2007), 767-785
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © J.UCS

A Visual Language for Animated Simulation

Vladimir O. Di Iorio
(Universidade Federal de Viçosa, Brazil
vladimir@dpi.ufv.br)

Débora P. Coura
(Centro Universitário do Leste de Minas Gerais, Brazil
dcoura@terra.com.br)

Leonardo V. S. Reis
(Universidade Federal de Viçosa, Brazil
leo@dpi.ufv.br)

Marcelo Oikawa
(Universidade Federal de Viçosa, Brazil
moikawa@dpi.ufv.br)

Carlos R. M. Junior
(Universidade Federal de Viçosa, Brazil
crmarques@dpi.ufv.br)

Abstract This paper presents a visual language for producing animated simulations. The language is implemented on a tool called *Tabajara Animator*, using principles of *Programming By Demonstration (PBD)*, which is a technique for teaching the computer new behaviour by demonstrating actions on concrete examples. The language is based on a formal model for concurrent update of agents, which represent the animated characters. The visual rules follow the "before-after" style, adopted by the most important similar tools. New features discussed by this work may produce a significant reduction on the number of required rules for producing animated simulations. This paper shows how these new features are implemented on a visual user-friendly interface, and how they are translated into structures of the formal model adopted.

Key Words: visual programming, programming by demonstration
Category: D.1.7

1 Introduction

Programming by Demonstration (PBD) is a technique for teaching the computer new behaviour by demonstrating actions on concrete examples [Lieberman 2001]. This technique has been used with success in several areas, for example, the construction of Web pages [Sugiura 2001], programming on Geographical Information Systems (GIS) [Traynor and Williams 2001], minimizing typing in small handheld devices [Masui 1998]. Tools for producing animated simulations represent an area with the first commercial systems that applied PBD successfully. Examples are *Stagecast Creator* [Smith et al. 2000] and *Agentsheets* [Repenning and Sumner 1995].

In [Coura et al. 2006], the authors propose three enhancements for PBD-based animated simulations systems. The most important enhancement is *first-person perspective* for visual rules. Stagecast Creator and Agentsheets use third-person perspective, leading to programs with several very similar visual rules, whose only difference is the orientation of the characters. The other new features are *negative conditions* and the use of *inheritance*. Using a significant example, it is shown that the enhancements may produce an important reduction on the number of visual rules required for a simulation. But important issues concerning the implementation of the enhancements are not solved.

This paper is an extension of the work published in [Coura et al. 2006], presenting the visual language of a system called *Tabajara Animator*. This system proposes visual solutions for the implementation of *negative conditions* and *inheritance*, and solves problems with the automatic generation of graphical representation of characters, associated with *first-person perspective*. The visual language is based on a formal model called ASM-OBJ, inspired by the Gurevich's *Abstract State Machines (ASM)* [Börger and Stärk 2003]. This paper also shows how visual programs containing the new features may be translated into programs of this model.

In Section 2, the most important tools for producing animation with PBD are presented. It is shown how the enhancements proposed in [Coura et al. 2006] may reduce the number of visual rules required, in situations commonly used. Section 3 introduces ASM-OBJ, a formal model for the definition of concurrent update of agents. In order to help explaining the semantics of the new visual language features, their translation into structures of the ASM-OBJ model is discussed, in the following sections. Section 4 presents an overview of the Tabajara Animator system. Sections 5, 6 and 7 analyzes the implementation of negative conditions, inheritance and first-person perspective for rules, respectively. In Section 8, the final conclusions are presented and future works are discussed.

2 Tools for Animated Simulation using PBD

Some of the most successful tools for animated simulation using programming by demonstration BD are *Stagecast Creator* [Smith 2000, Smith et al. 2000] and *Agentsheets* [Repenning and Sumner 1995]. They have similar features, but the visual rules in Stagecast Creator are called *visual before-after rules*, while in Agentsheets, they are called *graphical rewriting rules*.

The *Kidsim* project [Smith et al. 1994], later called *Cocoa*, was finally designated *Stagecast Creator*. It has been used mostly in children education. As

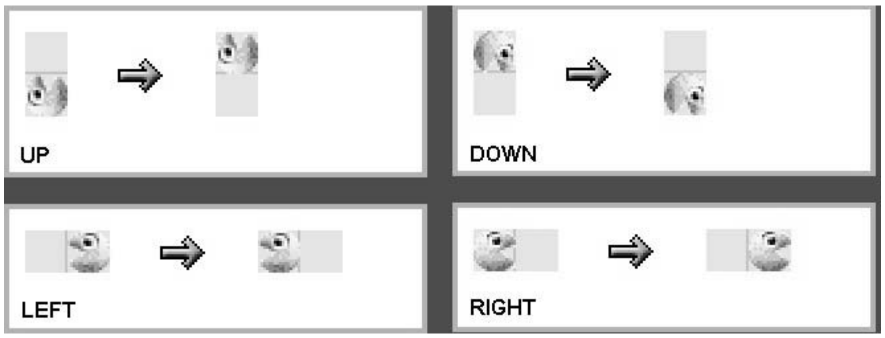


Figure 1: *Visual before-after rules* in Stagecast Creator.

the original name implies, it was intended to allow kids to construct their own simulations, reducing the programming task to something that anyone could handle.

Figure 1 shows four Stagecast Creator rules, taken from a version of the classic *Pacman* game [DeMaria and Wilson 2003]. The main character, the Pacman, is controlled by the user, using the keyboard. It moves in a labyrinth, together with ghosts. The rules on Figure 1 define the movement for the Pacman, when there is an empty space in front of it, in four different directions. The visual representation of the rules are separated in two parts, by a horizontal arrow. The left part, designated *before* clause, represents a possible situation occurred during a simulation. For example, the first rule represents a situation when the character is looking at a position on a cell over it, and this cell is empty. The right part of the rule, designated *after* clause, represents an action that must be executed when the condition of the *before* clause is satisfied. In order to define the *after* clause of the first rule, the user *demonstrates* his intention by moving the character up.

The Agentsheets environment aims to achieve a wide range of users, from children to professionals. The system includes a compiler which translates the visual rules to Java code, producing more efficient simulations than the ones built using Stagecast Creator.

Figure 2 shows a visual rule of Agentsheets with the same semantics as the first rule of Figure 1. In Agentsheets, to indicate that a character must move when an empty space is found, the user must define a position update on a graphical rewiriting rule. The condition to be satisfied defines that the character must have the given graphical representation, and the cell over it must be empty. The action executed when this condition is satisfied is a movement to a cell over the current position. Similarly to Stagecast Creator, additional rules must be built to define movements to other directions.

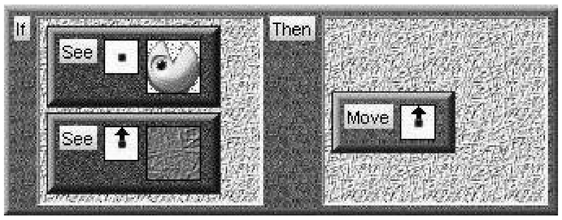


Figure 2: *Graphical rewriting rule* in Agentsheets.

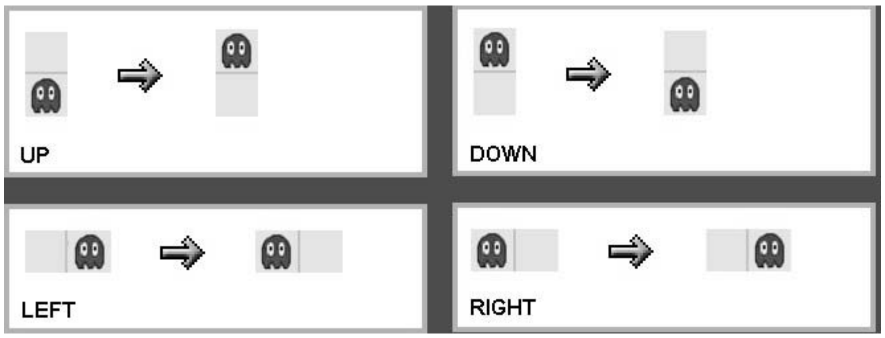


Figure 3: Rules for the movement of *ghost* characters in Stagecast Creator.

The Tabajara Animator project was created in order to implement the new features proposed in [Coura et al. 2006], for visual rules using PDB: *first-person perspective*, *negative conditions* and *inheritance*. The authors demonstrate that, with these innovations, the numbers of rules required for animated simulations can be significantly reduced.

In the examples of figures 1 and 2, as the character may be pointing to four different directions and the rules are written in third-person perspective, it is necessary to write four different rules to produce the desired behaviour. First-person perspective for rules is one of the new ideas implemented in Tabajara Animator. With first-person perspective, this behaviour can be defined by a single rule. Section 7 shows details of this important enhancement, and how it is implemented in Tabajara Animator.

In Stagecast Creator, the rules of Figure 3 may define the movement over empty spaces for a character representing a ghost, on the Pacman game. These rules are almost identical to the ones of Figure 1, the only difference is the character involved. This situation is a good opportunity to explore *inheritance*. The desired behaviour may be defined on a class named *Moveable*. Classes *Pac-man* and *Ghost* may be subclasses of *Moveable*, inheriting all the rules of the superclass. Section 6 discusses the problems involved in the implementation of

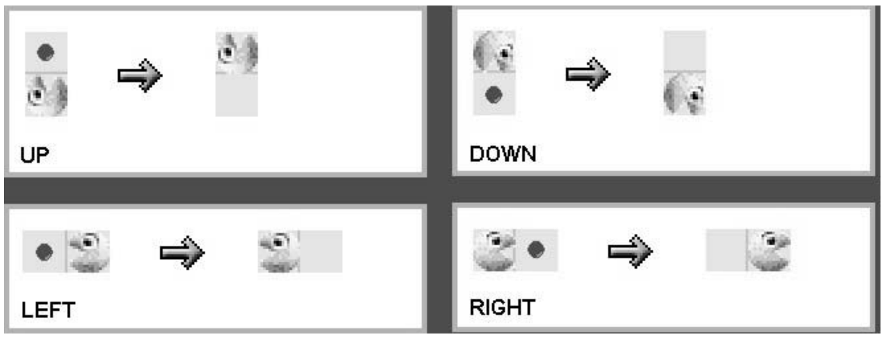


Figure 4: Movement of Pacman over vitamins, in Stagecast Creator.

inheritance.

Figure 4 shows the rules defining the movement over cells containing a "vitamin", for a character representing a Pacman. These rules are almost identical to the ones of Figure 1. The two sets of rules could be replaced by a rule with *negative condition*. The new rule could have the following semantics: the ghost must move to a cell in front of it, if this cell does **not** contain a wall of the labyrinth. Section 5 discusses the implementation of negative conditions.

3 The ASM-OBJ Formal Model

Abstract State Machines (ASM), formerly known as *Evolving Algebras*, are a formal model where the state of a system is represented by functions, and transitions are based on *function update*. A complete definition of this model can be found in [Börger and Stärk 2003]. ASM-OBJ is an extension of the ASM model which includes elements from object-oriented languages. Object-oriented extensions for ASM have been proposed in works like [Janneck and Kutter 1998] and [Zamulin 1998]. ASM-OBJ has been created especially for serving as basis for the Tabajara Animator visual language, and shares little similarities with these previous works.

This section presents an informal definition of the main elements of the model. The semantics is given by associations with pure ASM. The complete definition of ASM-OBJ can be found in [Coura 2006].

The syntax of ASM-OBJ is defined using a XML schema [van der Vlist 2002], so ASM-OBJ programs are well-formed XML documents. A reason for choosing XML instead of conventional syntax is that the visual language is intended to be shared by different applications. The code is automatically generated by visual tools, so it is not necessary to actually write programs using XML syntax, what would be an important drawback. In this section, parts of the ASM-OBJ model

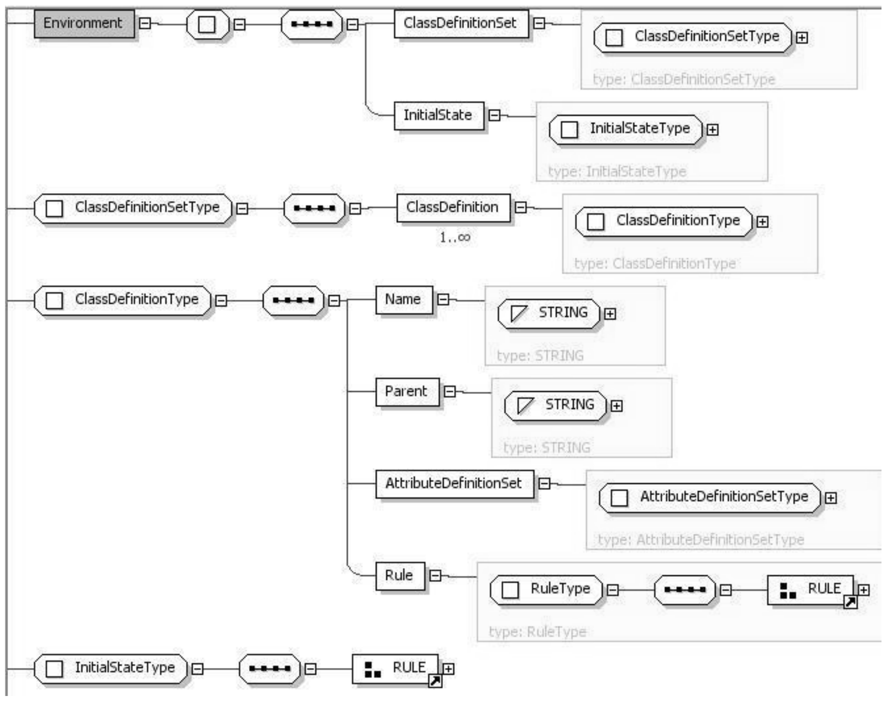


Figure 5: ASM-OBJ *environment* definition.

definition are presented using a visual representation for XML schemas, and the semantics is explained using pure ASM. The visual representation used is the one proposed by the *Oxygen* system [Wheller 2002], with icons like ▢ to represent a complex type, ▭ to represent composition and ⋮ to represent a choice.

Figure 5 shows the definition of an *environment* in ASM-OBJ, composed by a set of *class definitions* and an *initial state*. The definition of a class includes its name, the name of its parent on the hierarchy, a set of *attributes* and a *rule*. The initial state is defined by a *rule*. The semantics is explained in terms of the ASM model, as follows. Each ASM-OBJ class with name C is equivalent to an ASM *universe* with the same name, i.e., an unary relation identified by the set of elements e such that $C(e) = true$.

The definition of class attributes in ASM-OBJ, not shown in Figure 5, includes the name of the attribute and its type, which can be scalar types such as *Integer*, *Real* and *String*. Each type may be interpreted as another ASM universe. An attribute named A, inside a class named C, is interpreted as an ASM unary function $A : C \rightarrow T$, where T is the interpretation of the attribute type. An *agent* of a class named C acts like an *object* of OO languages, and is interpreted as an element of an ASM universe C.

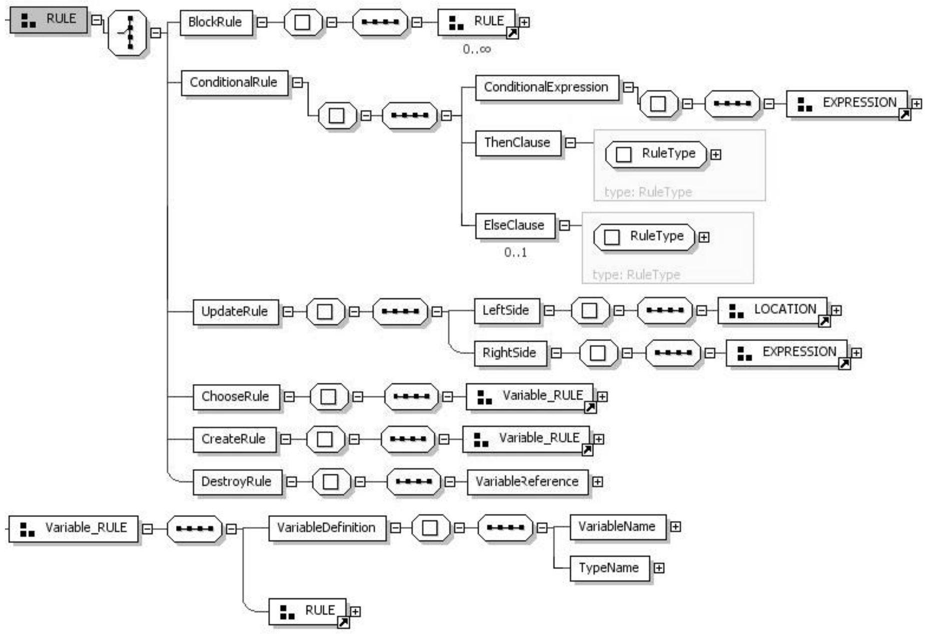


Figure 6: Some ASM-OBJ *rules*.

ASM-OBJ defines several types of rules, very similar to ASM rules. Figure 6 shows part of the definition of some of these rules. The semantics of the *execution* of an ASM-OBJ rule is equivalent to *firing* an ASM rule.

The execution of an ASM-OBJ *update* rule produces an update pair (*location*, *value*), just like in ASM, calculated using the *LeftSide* and *RightSide* components of the rule. A *location* defines an unique "address" which is associated to a value. In ASM, it is defined by a function and arguments. In ASM-OBJ, it is defined by an agent and an attribute of the agent´s class.

The result of executing a *block* is the union of the execution of each of its subrules. The result of executing a *conditional* rule is the execution of the *then* clause, when the *conditional expression* is satisfied; otherwise, it is the execution of the optional *else* clause.

In a *choose* rule with variable v of class C, an agent a of the class C is nondeterministically chosen and the result is the execution of the defined subrule, with the value of v set to a. In a *create* rule with variable v of class C, a new agent a of the class C is created and the result is the execution of its subrule, with the value of v set to a.

Inside the rule associated to a class, the reserved name *this* is an unbound variable. The *complete rule* of a class named C is the union of the rule associated to this class with the rules associated to all its superclasses, following the hierarchy defined by the environment. The execution of an agent a of a class C is the execution of the complete rule associated to the class C, with the value of the variable *this* set to a.

The execution of an environment starts with the execution of its initial state rule. This rule is usually a block containing *create* rules, which will build an initial set of *live agents*. Then, each step of the execution consists of selecting a subset of agents from the live agents set, executing the selected agents and finally building a new state. A new state is built applying the produced update pairs to the current state. As in pure ASM, applying an update pair (l, k) to a state produces a new state where the value associated to location l is replaced by k.

ASM-OBJ defines a rich set of standard functions which can be used in expressions. Another set of functions, called *external functions*, may be extended by users. External functions are used, primarily, for the communication with the external environment. Two calls to an external function, in different steps of an execution, may return different values, even when given the same arguments.

4 Overview of Tabajara Animator

Tabajara Animator is a system for the creation of animated simulations using programming by demonstration techniques. It is similar to Stagecast Creator and Agentsheets, but with the additional features proposed in [Coura et al. 2006].

To build visual programs, the system offers an *Hierarchy Editor*, which allows the definition of hierarchy relations, and a *Behaviour Editor*, which allows the definition of visual rules for each class. To present simulations, the system offers several different *animated simulation windows*. Simulations windows are discussed in Section 7.2.

Figure 7 shows a snapshot of the Hierarchy Editor window of Tabajara Animator. The user may create new classes anywhere in the hierarchy, defining a standard visual representation. In this example, *Brick*, *Moveable* and its two subclasses *Ghost* and *Pacman* are user-defined classes. The root of the hierarchy is the predefined class *VisibleClass*, representing any object with visual representation in an animated simulation. Figure 8 shows part of the *ClassDefinitionSet* element of an ASM-OBJ environment, representing these hierarchical relations.

Figure 9 shows the Behaviour Editor window, which can be activated from the Hierarchy Editor. After selecting a class and activating the Behaviour Editor, the user can create, delete or modify rules for this class. The visual rules have a strong correspondence with the structure of an ASM-OBJ program. Visual conditional rules are formed by a "before" clause, which represents the visual condition to

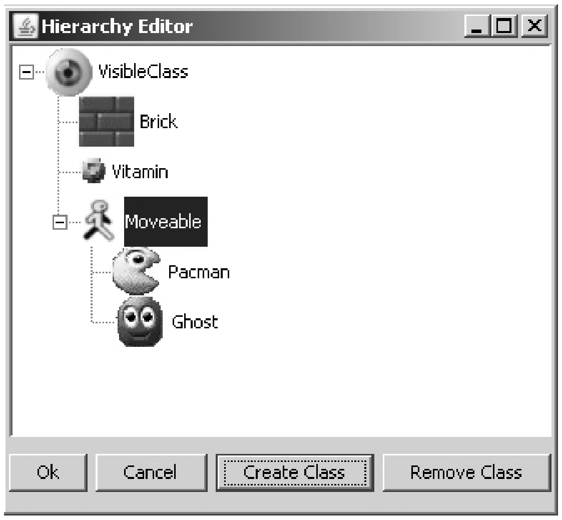


Figure 7: Hierarchy editor of Tabajara Animator.

```
...
<ClassDefinition>
  <Name>Moveable</Name>
  <Parent>VisibleClass</Parent>
  ...
</ClassDefinition>
<ClassDefinition>
  <Name>Pacman</Name>
  <Parent>Moveable</Parent>
  ...
</ClassDefinition>
...
```

Figure 8: Representation of Figure 7 in ASM-OBJ.

be evaluated, an “after” clause, which represents the rule to be executed if the condition is satisfied, and an optional “else” clause, which represents the rule to be executed if the condition is not satisfied. An “after” clause is usually a visual update rule. An “else” clause may be a visual update or another visual conditional rule.

Figure 10 shows a visual conditional rule for the Pacman class. “Before” and “after” clauses are separated by horizontal (green) arrows. “Else” clauses

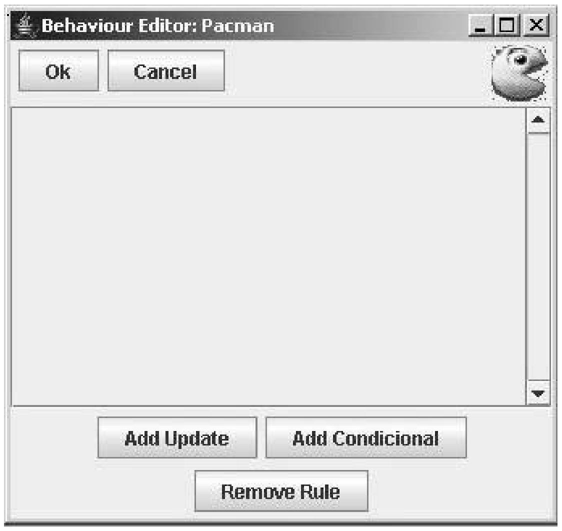


Figure 9: Behaviour Editor Window.

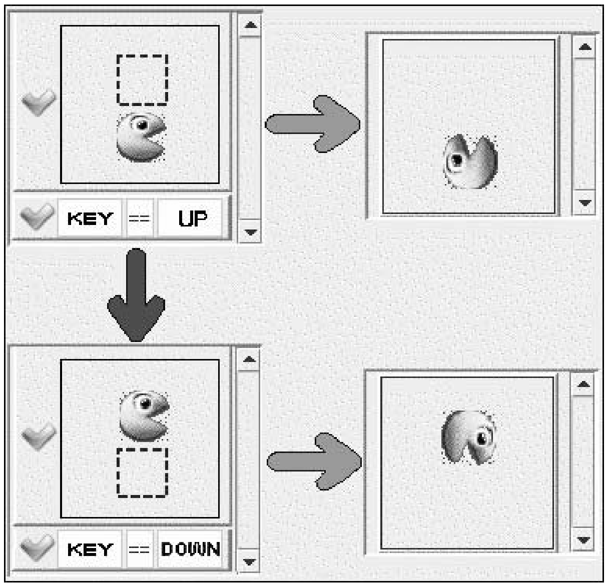


Figure 10: A complex visual conditional rule.

appear behind vertical (red) arrows. Conditions, inside "before" clauses, are visual predicates connected by an implicit logic operator *and*. For example, the first condition shown in Figure 10 is satisfied if there is no character above the Pacman, **and** the *up arrow* keyboard is pressed. Visual update rules may define modifications on the visual attributes of characters: their position, rotation and visual representation. In Figure 10, the user demonstrated the intended behaviour by defining visual update rules with a rotation applied to the graphical representation of the character. So, during a simulation, if the condition is satisfied, the character will be rotated as defined by the update rule.

Tabajara Animator offers a user-friendly interface to define visual conditional and update rules. The rectangular area with dashed sides, used in visual conditions, can be placed anywhere inside the window. If it is left empty, then the condition will be satisfied, during simulation time, only when there is no character inside the defined area. If characters are dragged into the rectangular area, then the condition is satisfied, during simulation time, when objects of the chosen classes are found inside the defined limits. Characters selected in "before" clauses appear automatically in "after" clauses, with the same position and rotation, unless the *negation operator* is applied to the visual condition (see Section 5). Moving and rotating these characters in "after" clauses, an user defines a visual update that will be executed on simulation time, when the visual condition is satisfied. Deleting characters or inserting new characters in "after" clauses, an user defines destruction and creation of characters, in simulation time.

An ASM-OBJ interpreter is integrated to the Tabajara Animator interface. Before carrying on simulations, the system translates the visual rules into ASM-OBJ code. The translation of visual elements discussed in this section is very obvious. Visual conditional rules are translated to ASM-OBJ conditional rules, with visual "before" clauses associated to conditional expressions, "after" clauses associated to *then* clauses and visual "else" clauses associated to ASM-OBJ *else* clauses. Complex visual conditions, with more than one predicate, are implemented using a call to an ASM-OBJ standard boolean function *and*. Visual update rules are translated to ASM-OBJ update rules, with class attributes representing position, rotation and other visual attributes.

5 Negative Conditions

The structure of a visual conditional rule is itself a way to define negative conditions. Any rule inside an "else" clause will be executed only if the condition is **not** satisfied.

But the system offers also a visual *negation operator* which can be associated to any predicate. It indicates that a condition is true only if the predicate is

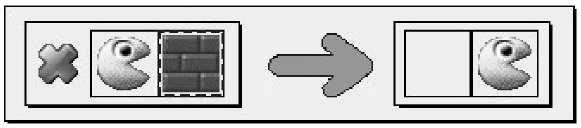


Figure 11: A rule with negative condition.

not satisfied. The negation operation, together with the implicit *and* operator described in Section 4, allows the definition of complex conditions. In systems like Stagecast Creator and Agentsheets, users are restricted to a more limited set of possible conditions.

Figure 11 shows an example of a rule with negative condition, for the Pacman class. A symbol ✓ appears before every positive condition. If the user wants a negative condition, this symbol must be replaced by a symbol ✖ The interface allows the user to change from positive to negative conditions, and vice-versa, with a click of the mouse. Using also the concept of first-person perspective, the semantics of the rule shown is: if the character is looking at a cell which does **not** contain a brick, then the character is moved to this cell. The visual negation operator is straightforward translated to ASM-OBJ by using a call to a standard boolean function *not*.

6 Inheritance

In [Repenning and Perrone 2000], the authors present some reasons not to use inheritance in systems for the creation of animated simulations with visual languages. They argue that abstractions like inheritance are nontrivial for end users to understand, and are hard to represent visually. They present a different approach for generalization, called *programming by analogous examples*.

Examples presented in [Coura et al. 2006] show that inheritance may indeed reduce the number of required rules in some simulations. The authors agree that it may be an abstract concept not very easy to understand by end users, but they argue that a system may offer inheritance as an additional feature, reserved for more advanced users.

This work shows how Tabajara Animator solves the problem of representing inheritance relations and inherited rules on subclasses. As in many other systems, hierarchical relations are visually represented by a tree diagram, as shown in Figure 7. The class associated with a tree node is the superclass of the classes associated with the siblings of this node. For example, *Pacman* is a subclass of *Moveable*. Figure 12 shows two instances of the Behaviour Editor window. The window associated to the Moveable class shows a rule with negative condition. The window associated with the Pacman class shows the inherited rule, changing

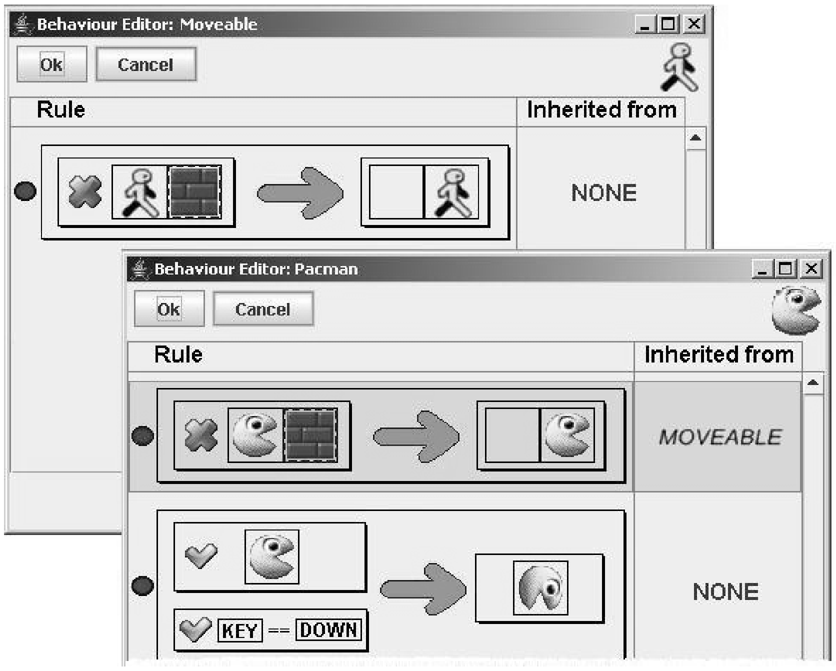


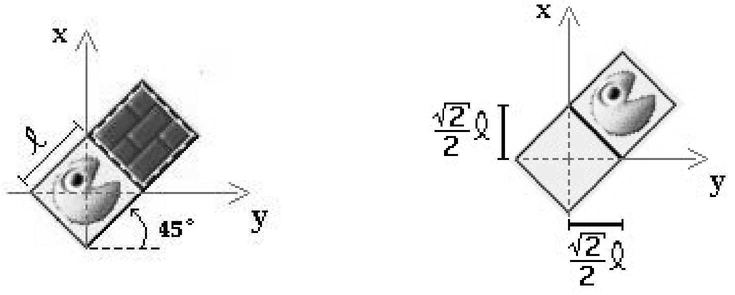
Figure 12: Representing inherited rules in subclasses.

the graphical representation of the character. With this solution, it is easy to understand the behaviour of a class, even when it has inherited rules. But the system allows the edition of rules only on the classes where they are originally defined. On subclasses, inherited rules are “read-only”.

7 First-Person Perspective

Rules in Stagecast Creator and Agentsheets use third-person perspective. As the examples of Section 2 show, this strategy may lead to visual programs with several similar rules, where the only difference is the orientation of the characters. In order to implement first-person perspective for rules, it is not necessary to add new features to the interface. However, a lot of additional work must be done by the system, as discussed below.

The rule represented in Figure 11, when using first-person perspective, may be equivalent to several third-person perspective rules. The results produced in



(a) Verify Brick on a cell. (b) Move the character.

Figure 13: Calculations needed to implement first-person perspective.

simulation time depend on the character current rotation. For example, suppose that the character is rotated 45 degrees counterclockwise. The real position of the rectangular area with dashed sides, where a Brick character may be positioned, must be calculated as shown in Figure 13(a). The update defined by the rule of Figure 11 represents only a horizontal movement. But in simulation time, this movement may have horizontal and vertical components, which must also be automatically calculated, as shown in Figure 13(b). All these calculations are carried out by ASM-OBJ code properly generated.

7.1 Code generated for managing first-person perspective features

Tabajara Animator defines *VisibleClass* as the superclass of any visible character. The attributes of VisibleClass define the current graphical representation, the current rotation and the horizontal and vertical components of the current position.

A visual conditional rule including a rectangular area with dashed sides is translated into an ASM-OBJ *choose* rule. The translation of the visual condition defined in Figure 11 is the ASM-OBJ *choose* rule shown in Figure 14. This rule defines a variable named x, whose type is the class named *Brick*. Its execution forces the system to find an agent of the Brick class, satisfying conditions defined in the subrule. The subrule of the *choose* rule, in this case, is a *conditional* rule. The condition to be evaluated is the result of the application of the standard boolean function *not* to a call to an *external function* named *RectangularArea*.

The *RectangularArea* function has a boolean return type. Arguments passed to this function, not shown in Figure 14, represent the base object, the position of the rectangular area and a variable passed by reference, which may be instantiated by the function. The base object, in this case, is the Pacman character. The position of the rectangular area is defined by its top left and right down corners, relative to the position of the base object. These values are translated into

```
<ChooseRule>
 <VariableDefinition>
  <VariableName>x</VariableName>
  <TypeName>Brick</TypeName>
 </VariableDefinition>
 <ConditionalRule>
  <ConditionalExpression>
   <StandardFunctionCall>
    <FuncName>not</FuncName>
    <ArgumentList>
     <ExternalFunctionCall>
      <Name>RectangularArea</Name>
      <ParameterMap>...<ParameterMap>
     </ExternalFunctionCall>
    </ArgumentList>
   </StandardFunctionCall>
  </ConditionalExpression>
  <ThenClause> ... </ThenClause>
 </ConditionalRule>
</ChooseRule>
```

Figure 14: Rule of Figure 11, translated into ASM-OBJ.

```
<StandardFunctionCall>
 <FuncName>add</FuncName>
 <ArgumentList>
  <Location>
   <VariableRef>this</VariableRef>
   <AttribName>posX</AttribName>
  </Location>
  <StandardFunctionCall>
   <FuncName>Xproj</FuncName>
   <ArgumentList>
    <IntegerConst>30</IntegerConst>
    <IntegerConst>0</IntegerConst>
    <Location>
     <VariableRef>this</VariableRef>
     <AttribName>rotation</AttribName>
    </Location>
   </ArgumentList>
  </StandardFunctionCall>
 </ArgumentList>
</StandardFunctionCall>
```

Figure 15: Piece of ASM-OBJ program representing horizontal position update.

real coordinates using the current position and the current rotation of the base character, as shown in Figure 13(a). Finally, the last argument is the variable x, bound by the *choose* rule. External functions receive an implicit argument that allows them to access the whole environment, during a simulation. Using this implicit argument and the arguments explained above, the *RectangularArea* function scans the environment and verify whether there is a Brick character inside the calculated area. If such a character is found, the value of variable x is set to it, and the function returns *true*. Otherwise, the function returns *false*.

As shown in Figure 13(b), a position update requires the calculation of horizontal and vertical displacements. To solve this, every position update is translated into two ASM-OBJ update rules, using predefined functions named *projX* and *projY*. Figure 15 shows one of the update rules resulting from the translation of the position update defined in Figure 11. To the value of the *posX* attribute of the character, which represents the horizontal component of the current position, the rule adds the result of a call to the *projX* function. The *projX* function receives three parameters: a horizontal displacement, a vertical displacement and a value associated with the rotation of the character. The result is the *real* displacement on the horizontal axis. The code shown in Figure 15 supposes that the graphical representation of the Pacman character is a square with side length of 30 pixels. So the call to the *projX* function calculates the **horizontal** result of a horizontal movement of 30 pixels to the right, considering the current rotation of the character. The second required update rule is not shown, but it is similar to the code shown n Figure 15, replacing *posX* by *posY* and *projX* by *projY*. In this case, the call to the *projY* function calculates the **vertical** result of a horizontal movement of 30 pixels to the right, considering the current rotation of the character.

7.2 Problems in animated simulation windows

Rules in Tabajara Animator are defined using first-person perspective. But the system provides variations of first-person and third-person perspective animated simulation windows. On windows that present animated simulations with first-person perspective, a character is chosen to be positioned in the center of the window, and its graphical representation never changes. When this character suffers rotation, the window presents an inverse rotation of all other characters. In third-person perspective animated simulation windows, rotated graphical representations are automatically generated for each character, according to its rotation attribute.

A problem occurs in third-person simulation windows. Frequently, the graphical representations automatically generated for characters are not satisfactory. For example, suppose that the graphical representation of the Pacman, used on the rule of Figure 11, is rotated 180 degrees clockwise (or counterclockwise).

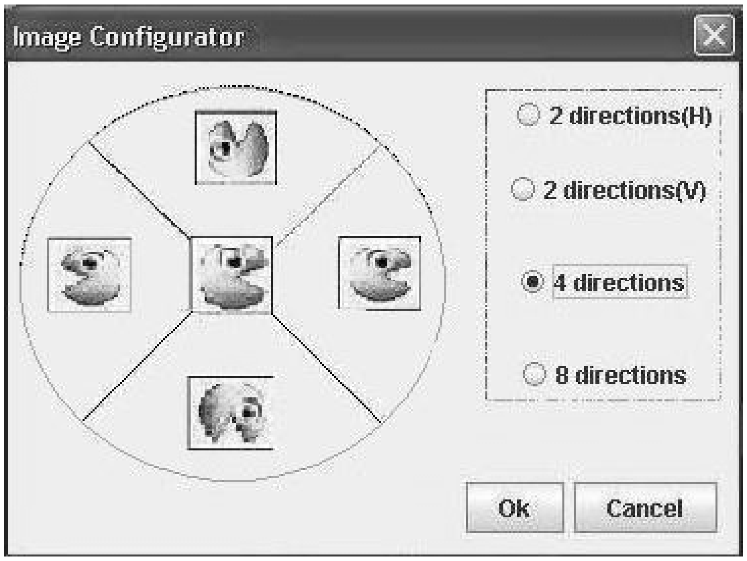


Figure 16: Defining graphical representation, depending on rotation.

The resulting picture would be an upside down character. In order to solve this problem, Tabajara Animator offers a visual tool called *Image Configurator* which allows users to define different graphical representations for characters, depending on their rotation attribute. An example is shown in Figure 16, with four different graphical representations defined for the Pacman character. Each different image is associated with a range of values for the rotation attribute of a Pacman character. During simulation time, the system verifies the rotation of each agent of the Pacman class and shows the corresponding image.

The system translates the operations defined in an Image Configurator to ASM-OBJ code. When using third-person perspective animated simulation windows, an additional ASM-OBJ conditional rule is generated. This rule updates the attribute defining the graphical representation of the character, according to its rotation attribute.

8 Conclusions and Future Works

In [Coura et al. 2006], inheritance, negative conditions and first-person perspective for rules are enhancements proposed for PBD-based animated simulation tools. An example of a simple application with common situations found in animated simulations is defined. The application is implemented in Stagecast Creator and Agentsheets, which are tools with third-person perspective rules, without inheritance and with little support for negative conditions. The number of rules required for these implementations is compared to a tool with full support for the proposed enhancements. The results show that an important reduction on the number of rules may be achieved, using the enhancements.

The results presented in [Coura et al. 2006] are very significant, but they depend on an a successful implementation of all the proposed features. The main contribution of the work presented in this paper is the confirmation that inheritance, negative conditions and first-person perspective for rules may be successfully implemented. The paper shows details of the implementation, on a system called *Tabajara Animator*, using the help of a formal model to explain the semantics of the visual elements.

The implementation of the proposed features, in Tabajara Animator, may be summarized as follows. The interface of the system introduces new visual elements, not present in similar tools, in order to define negative conditions and inheritance. A visual negation operator may be applied individually to any predicate on a visual condition. It allows users to build more complex conditions than the ones provided by similar tools. Inheritance relations are represented on a tree diagram, and the problem of representing rules in subclasses is elegantly solved. First-person perspective for rules requires no additional visual elements on the interface of the system. The calculation of the real position of characters during simulation is carried out by predefined ASM-OBJ functions. An user-friendly visual tool allows the definition of different graphical representations for characters, solving the problems with automatically generated representations, in animated simulation windows.

The visual language of Tabajara Animator may be extended in several ways. An important extension may be the use of *rule abstractions*. The visual language of Agentsheets, called *Visual Agent Talk (VAT)*, offers such abstractions. A VAT *method* is a set of rules, which can be referenced by the name, inside other visual rules. An interesting work would be the implementation of a similar feature in Tabajara Animator. Because of inheritance, it would be necessary to define the behaviour in cases where a class has a method with the same name as a method defined in its superclass. Another possible extension to the Tabajara Animator visual language is the use of polymorphism. This feature has never been discussed together with languages for visual animation.

Acknowledgements

This work is partially supported by FAPEMIG, Brazil.

References

[Börger and Stärk 2003] Börger, E. and Stärk, R. (2003). *Abstract State Machines: A Method for High-Level System Design and Analysis.* Springer-Verlag.

[Coura et al. 2006] Coura, D., Di Iorio, V., Lima, A., Oliveira, A., and Andrade, M. V. (2006). Animações Através de Programação por Demonstração. In *Anais do Simpósio de Fatores Humanos em Sistemas Computacionais (IHC 2006)*, pages 81–90, Natal, Brazil.

[Coura 2006] Coura, D. P. (2006). Produzindo Animação Através da Programação por Demonstração. Master's thesis, Universidade Federal de Viçosa, Viçosa, Brasil.

[DeMaria and Wilson 2003] DeMaria, R. and Wilson, J. L. (2003). *High Score!: The Illustrated History of Electronic Games.* McGraw-Hill Osborne Media.

[Janneck and Kutter 1998] Janneck, J. and Kutter, P. (1998). Object-based Abstract State Machines. TIK-Report 47, Swiss Federal Institute of Technology (ETH) Zurich.

[Lieberman 2001] Lieberman, H., editor (2001). *Your Wish is My Command: Programming by Example.* Morgan Kaufmann.

[Masui 1998] Masui, T. (1998). Integrating Pen Operations for Composition by Example. In *ACM Symposium on User Interface Software and Technology*, pages 211–212.

[Repenning and Perrone 2000] Repenning, A. and Perrone, C. (2000). Programming by example: programming by analogous examples. *Commun. ACM*, 43(3):90–97.

[Repenning and Sumner 1995] Repenning, A. and Sumner, T. (1995). Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer*, 28(3):17–25.

[Smith 2000] Smith, D. C. (2000). Building personal tools by programming. *Communications of the ACM*, 43(8):92–95.

[Smith et al. 1994] Smith, D. C., Cypher, A., and Spohrer, J. (1994). KidSim: Programming Agents Without a Programming Language. *Communications of the ACM*, 37(7):54–67.

[Smith et al. 2000] Smith, D. C., Cypher, A., and Tesler, L. (2000). Programming by example: novice programming comes of age. *Commun. ACM*, 43(3):75–81.

[Sugiura 2001] Sugiura, A. (2001). Web Browsing by Demonstration. In Lieberman, H., editor, *Your Wish is My Command: Programming by Example*, pages 61–86. Morgan Kaufmann.

[Traynor and Williams 2001] Traynor, C. and Williams, M. G. (2001). End users and GIS: a demonstration is worth a thousand words. In *Your wish is my command: programming by example*, pages 115–134. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[van der Vlist 2002] van der Vlist, E. (2002). *XML Schema - The W3C's Object-Oriented Descriptions for XML.* O'Reilly.

[Wheller 2002] Wheller, S. (2002). <oXygen/> User Manual. SyncRO Soft Ltd. (retrieved 25 Jan, 2007, from http://www.oxygenxml.com/).

[Zamulin 1998] Zamulin, A. (1998). Object-oriented Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science.* Technical Report, Magdeburg University.